A Lightweight Framework for High-Quality Code Generation

Mohammed Latif Siddiq, Beatrice Casey, and Joanna C. S. Santos

University of Notre Dame, Notre Dame, IN, USA 46556

msiddiq3@nd.edu, bcasey6@nd.edu, joannacss@nd.edu

Abstract—In recent years, the use of automated source code generation utilizing transformer-based generative models has expanded, and these models can generate functional code according to the requirements of the developers. However, recent research revealed that these automatically generated source codes can contain vulnerabilities and other quality issues. Despite researchers' and practitioners' attempts to enhance code generation models, retraining and fine-tuning large language models is time-consuming and resource-intensive. Thus, we describe FRANC, a lightweight framework for recommending more secure and high-quality source code derived from transformer-based code generation models. FRANC includes a static filter to make the generated code compilable with heuristics and a quality-aware ranker to sort the code snippets based on a quality score. Moreover, the framework uses prompt engineering to fix persistent quality issues. We evaluated the framework with five Python and Java code generation models and six prompt datasets, including a newly created one in this work (SOEVAL). The static filter improves 9% to 46% Java suggestions and 10% to 43% Python suggestions regarding compilability. The average improvement over the NDCG@10 score for the ranking system is 0.0763, and the repairing techniques repair the highest 80% of prompts. FRANC takes, on average, 1.98 seconds for Java; for Python, it takes 0.08 seconds.

Index Terms—code generation, code quality, code security, code generation reliability, large language models

I. INTRODUCTION

Code generation techniques based on machine learning automatically produce source code from *prompts* [1] provided as input. These prompts provide a high-level specification of the developers' intent. Prompts can be single/multi-line comments, code expressions (*e.g.*, a function definition), or a combination of both. With the recent release of GitHub Copilot [2] and ChatGPT [3], ML-based source code generation tools are increasingly being used by developers in order to reduce software development efforts [4].

The automated code generation process is a *sequence-to-sequence* translation task, and Natural Language Processing (NLP) techniques can help tackle this task. With the rise of transformer architecture [5] and Large Language Models (LLMs), prior works focused on automating source code generation using these ML techniques [6]–[8]. Specifically, LLMs are trained and fine-tuned with *large* amounts of *code* snippets, including natural text, to understand the context and problem statements and generated code [9].

Although ML-based code generation techniques can produce functionally correct code, they may contain code smells and vulnerabilities [10]–[12]. A recent study showed that LLMs are fine-tuned with samples containing harmful coding patterns that leak to the generated code [10]. Another study found that GitHub Copilot can produce vulnerable code [11]. With the increasing use of LLM-based code assistants, these problematic code snippets can get deployed into production, negatively affecting the software system's security and reliability.

To improve the quality of the generated code, we first need a *good dataset* (*i.e.*, free of quality issues), but this is challenging because collecting training data is *time-consuming* [13] and open-source code commonly used for training *contain quality issues* (*e.g.*, bugs, vulnerabilities, and smells [10], [14], [15]). Moreover, fine-tuning an LLM model is a *resource-hungry* process [16]. Fine-tuning an LLM requires at least one GPU, and pre-training is typically performed on a large cluster of GPUs [16]. Although models can make inferences without GPUs, the throughput may not be optimal. For example, we used a GPU with 24 GB RAM for an inference model with 2.7 billion parameters to generate 128 tokens with up to 2,048 context tokens. To generate more tokens and infer larger models, more GPUs or a more expensive GPU with a larger RAM would be needed.

Although LLM fine-tuning can be done on the cloud (via an API) to avoid acquiring expensive GPUs, it is still costly. For example, OpenAI's fine-tuning API currently costs U$0.03 dollars per 1,000 tokens (8K context GPT-4 model) [17]. Since there can be billions of tokens in a large dataset, fine-tuning this model would cost thousands of dollars. Besides *fine-tuning* costs, there is a separate cost for *inference*. It costs U$0.12 per 1,000 tokens to use your own fine-tuned model.

Since LLMs can generate code with quality issues, we need a non-expensive way to provide the best-generated code to the user. In light of this need, this paper describes FRANC, a *lightweight*, *configurable*, and *model-agnostic* framework to *filter*, *rank*, and *repair* code automatically generated by LLMs. FRANC works by taking as input a developer's prompt and then using (i) static filters to remove non-compilable code (*filtering phase*), (ii) off-the-shelf quality issues detection tools to rank generated code snippets with respect to their measured quality (*ranking phase*), and (iii) prompt engineering to fix quality issues (*repairing phase*). To our knowledge, this is

the first framework for ranking and repairing generated source codes based on their measured quality.

To demonstrate FRANC's effectiveness, we conducted an empirical evaluation in which we used FRANC to filter, rank, and fix Java and Python code automatically generated by five models (CodeParrot, InCoder, CodeGen, PolyCoder, and ChatGPT). In this experiment, we generated code from **1,081** prompts collected from existing code generation benchmarks [9], [18]–[21] and **70** prompts we created from questions posted on StackOverflow.

This paper's contributions are **(1)** a novel framework (FRANC) to filter, rank and repair the output of code generation models based on code quality; **(2)** automated filtering capabilities to remove non-compilable and unnecessary portions of the generated code to minimize human inspection; **(3)** a demonstration of how prompt engineering (and different prompt repair structures) can help to repair quality issues automatically; **(4)** an empirical investigation comparing the effectiveness of the framework with existing code generation and infilling models; **(5)** a dataset of **70** prompts. This paper's replication package is available at: https://doi.org/10.5281/zenodo.8015394.

II. BACKGROUND

This section explains key concepts used in this work.

A. ML-based Code Generation and Infilling

ML-based code generation leverages NLP techniques to generate source code from a given context. The context can be a combination of natural language and source code. It can also include file/variable names, other files in the software system, *etc.* The code generation problem was previously tackled as a *sequence-to-sequence (seq2seq) learning* problem [22]. Prior works used Recurrent Neural Networks (RNN) and neural networks based on Long Short-Term Memory (LSTM) [22], [23] to generate source code. To memorize specific portions of the training input for learning, the LSTM and RNN use a feedback loop during training.

The attention-based transformer architecture revolutionized the field of language learning in 2017 [5]. The transformer is based on an encoder-decoder architecture that leverages the self-attention mechanism to weigh the importance of each input data point [5]. There are several transformer-based deep learning models, like BERT (Bidirectional Encoder Representations from Transformers) [24], T5 (Text-to-Text Transformer) [25] and GPT-3 (Generative Pre-trained Transformer) [26]. These language learning models can be fine-tuned with code-related datasets for source code completion [27]–[29], search [30], and summarization [31]. Examples of this type of model include CodeBERT [30], CodeT5 [32], and Codex [9].

Transformer-based code generation techniques use left-to-right (causal) autoregressive language modeling objectives [9], [26] or, as BERT does [24], [30], use masked language modeling objectives. That means that the generative model will take context from the *left* side of the cursor and will not take any

context *after* the cursor to generate source code. However, developers edit earlier parts of the code, and this edit action may depend on both sides of the cursor. Generating code by taking context from *both* sides is referred to as **code infilling**. InCoder [33] is an example of a code-infilling model.

B. Code Quality and Code Smells

Code quality is a broad term used to refer to code snippets that are free of bugs and conformant to its requirements [34]. The quality of a source code is usually expressed in terms of *defect rate* (*i.e.*, measured as the number of defects per unit, *e.g.*, lines of code, function points, *etc.*) and *reliability* (measured in terms of failure occurrences, *e.g.*, the number of failures during a period, mean time to failure, *etc.*). However, the main indicator of high-quality code that complies with specifications depends on the vendors' demands. To ensure code quality, a common **coding standard** needs to be adopted by every contributor. Different languages adopt specific coding practice protocols. For example, PEP-8 [35] is a well-known guide for coding practice standards for Python. It provides an extensive guide for code layout, whitespace usage, and naming conventions *etc.* For instance, according to the guide, the coding layout should have 4 spaces per indentation level, and spaces are the preferred indentation method.

An indication of poor system design and implementation practices is a **code smell** (also known as a "bad code smell" or "smell") [36]. These code smells can introduce software maintenance problems. Furthermore, they contravene fundamental software design rules, reducing the product's potential effectiveness. These issues increase the possibility of future errors or failures or can hinder software development [37], affecting the software's reliability. For example, the code in Listing 1 may throw a `ValueError` if the input at line 3 cannot be parsed to `int`. Although there is a `try-except` block to catch the exception (highlighted), the exception is not handled, which is an example of a code smell [38].

```
        ── Code smell example ──                  ── Security smell example ──
1  try:                              1  import hashlib
2      num = input('Enter number:')  2  def validate(c, h):
3      num = int(num)                3      hash_md5 = hashlib.md5(c)
4  except ValueError:                4      hash = hash_md5.hexdigest()
5      pass                          5      return hash == h
```

Listing 1: Examples of a code smell and security smell

Insecure coding, flaws with design choices, and coding standard violations all fall under the umbrella phrase "code smell". **Security code smells** (or simply "security smells") are a *subset* of code smells. They have frequently used programming patterns that could result in vulnerabilities [39], [40]. Security smells point to the possibility of a vulnerability, even if they may not constitute vulnerabilities entirely by themselves [41]. The code (shown on the right) in Listing 1 is taken from CodeQL examples [42]. It uses the `md5` hash function that is unsafe, which is related to *CWE-327: Use of a Broken or Risky Cryptographic Algorithm* [10].

In our work, we focus on the **quality of automatically generated code** with respect to following *code standards*, as well as the absence of *code smells* and *security smells*.
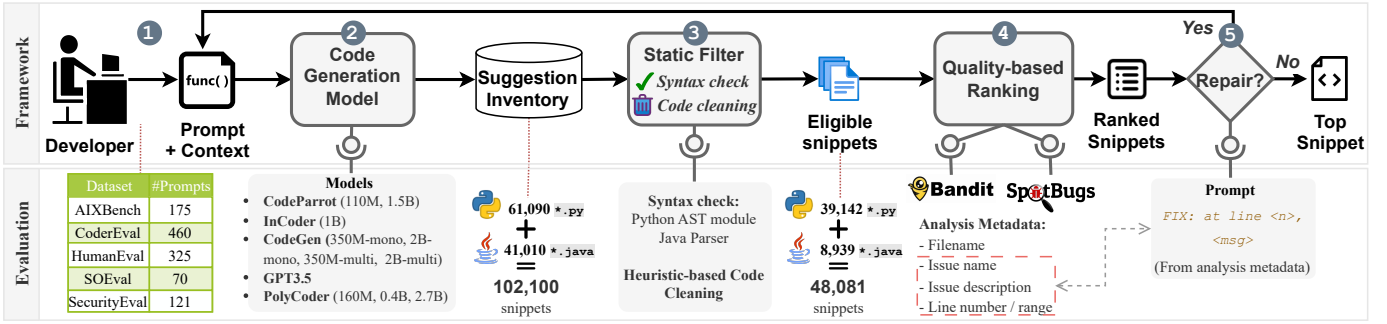
Fig. 1. Framework overview and evaluation parameters

C. Motivating Example

A code generation model can generate multiple (*ranked*) suggestions for a given prompt. However, some produced suggestions may contain *quality issues* (*e.g.*, security smells) [10]. For example, consider that we provide the prompt below (highlighted lines) to GitHub Copilot [2]. The generated code (lines 6-9) on the first position of GitHub Copilot's rank is *functionally correct* but contains a *SQL injection vulnerability*. It uses a formatted string to construct the query (line 7).

```
_____ utils/users.py (1st generated suggestion) _____
1  from django.db import connection
2  def show_user(request, username):
3      '''
4      create a cursor to get user info from users table and return it
5      '''
6      cursor = connection.cursor()
7      cursor.execute("SELECT * FROM users WHERE username = '%s'" % username)
8      user = cursor.fetchone()
9      return user
```

When we inspect the remaining generated suggestions, we can see code snippets that do not contain a SQL Injection. One of the lowest-ranked suggestions (shown below) is using a parameterized query to avoid SQL injection vulnerability.

```
_____ utils/users.py (7th generated suggestion) _____
1  from django.db import connection
2  def show_user(request, username):
3      '''
4      create a cursor to get user info from users table and return it
5      '''
6      cursor = connection.cursor()
7      cursor.execute("SELECT * FROM users WHERE username = %s", [username])
8      user = cursor.fetchone()
9      return user
```

Since the previous suggestion is not the top one recommended generated by the model, we need a lightweight ranking system to address the quality issues of the generated suggestions. Thus, our goal is to show as the top suggestion to the user a source code that is vulnerability-free, smell-free code, and follows standard coding practices.

III. OUR FRAMEWORK (FRANC)

Fig. 1 provides an overview of our lightweight quality-aware framework for code generation (*"Framework"* swimlane). Our framework has five phases that are ***model-agnostic***. Each phase can be tailored to the underlying programming language and LLM being used by the developer (as shown in the *"Evaluation"* swimlane).

In the ❶ ***prompt and context creation*** phase, software engineers specify the expected functional behavior of the code to be generated along with its context. ❷ In the ***code generation***

phase, FRANC uses an existing LLM to generate code snippets. Since these models may generate an n amount of sorted samples, the output of this phase is the framework's *suggestion inventory* (*i.e.*, a sorted set of n automatically generated code snippets). ❸ In the lightweight ***static filtering*** phase, FRANC applies heuristics to generated code in order to automatically fix and/or remove code snippets in the suggestion inventory with syntax errors. The output of this phase is a set of x *eligible code snippets* that passed the filtering criteria (where $x \leq n$). ❹ In the ***quality-based ranking*** phase, FRANC sorts the code snippets from the previous phase based on a *configurable* quality score. ❺ In the ***repairing*** phase, FRANC automatically repairs problematic generated code via prompt engineering. This phase involves the insertion of a prompt that instructs the code generation technique to fix a specific problem (*e.g.*, *"Fix the buffer overflow in line 10"*). These five phases are detailed in the next sections.

A. Phase 1: Prompt Creation

In this first phase, the software engineer creates a *prompt* with a surrounding *context* for the code generation technique. The lines 1–6 (highlighted in blue) in the code snippet in Listing 2 is an example of input an engineer can provide to an LLM. It contains an import statement, a function declaration, and a comment describing the function's intended functionality. This prompt instructs the LLM to generate the body of the function yaml_load(filename) that will parse a YAML file, create an object with the loaded data, and return it to the caller of the function. The LLM may also use the filename (utils/parser.py) as the context.

B. Phase 2: Code Generation

FRANC uses an LLM to generate code. Since our framework is model-agnostic, the model can be an open-source model (*i.e.*, datasets and parameters are publicly available) or a closed-source one (commercial). The model takes as input the *prompt* and surrounding *context* and produces multiple sorted code suggestions (snippets). The model's sorted list of code snippets composes FRANC's ***suggestion inventory***. For instance, the code in Listing 2 is the top suggestion generated by CodeGen (2 billion parameters) [43] when given the prompt described in Section III-A. This code repeats the prompt (lines 1-6),

provides an implementation to `yaml_load(filename)`, and contains an *unnecessary* and *incomplete* function (lines 10-19). This incomplete function serializes objects into a YAML file, which does not match the intent in the prompt and has a syntax error (missing closing parenthesis at line 19). Thus, in the next phase, FRANC removes this unnecessary function and other types of syntax issues.

```
                            ─── utils/parser.py ───
1  import yaml
2  def yaml_load(filename):
3      '''
4      Use the filename variable to open a file, load the contents into a variable,
5      and return it
6      '''
7      with open(filename) as f:
8          return yaml.load(f, Loader=yaml.FullLoader)
9
10 def yaml_dump(data, filename, full=True):
11     '''
12     take data variable, filename variable and boolean
13     variable full, write the contents of data variable
14     to a YAML file with the filename variable
15     '''
16     with open(filename, 'w') as f:
17         if full:
18             return yaml.dump(data, f, sort_keys=False,\
19             width=1000, allow_unicode
```

Listing 2: Generated code example

C. Phase 3: Static Filtering

As shown in Listing 2, code snippets in the suggestion inventory may include unnecessary blocks of code (*i.e.*, lines of code that do not match the intent specified in the prompt). Moreover, they may have syntax errors [10]. Consequently, developers need to manually vet and fix the code when it is not compilable or contains unnecessary extra code. Therefore, in this third phase, our framework inspects each snippet from the *Suggestion Inventory* and applies a set of **heuristics** to *(a)* remove unnecessary code from the snippet, and *(b)* check whether it is syntactically correct. The output of this phase is a set of **eligible snippets** that had any extra code removed from it **(a)** and passed the syntax check **(b)**.

The heuristics used for this filtering phase are *configurable*, meaning that they are tailored to the underlying programming language being used. For example, a heuristic that can be applied to clean the Python code in Listing 2 is to remove any code *after* from the prompt's function. The resulting snippet would not contain lines 10-19.

D. Phase 4: Quality-based Ranking

Automatically generated code can contain several quality issues, such as coding standard violations, code smells, and vulnerabilities [10], [11], [21]. Thus, the ranking used by the code generation model may produce code with quality issues as the first (top-1) suggestion. Hence, FRANC includes a configurable *quality-based ranker* that sorts code snippets in a model's output based on a **quality score** $\mathbf{Q(c_i)}$:

$$Q(c_i) = \sum_{j=1}^{m} w_j q_j(c_i) \text{ where } \sum_{j=1}^{m} w_j = 1 \text{ and } w_j \geq 0 \quad (1)$$

Each *code snippet* c_i ranked at the position i in the model's output is evaluated according to different *quality factors* $q_j(c_i)$ that take into account a specific quality attribute (*e.g.*, security, performance, code smells *etc.*). Each *quality factor*

$(0 \leq q_j(c_i) \leq 1)$ has a corresponding non-negative *weight* w_j. Thus, the quality score $Q(c_i)$ is a weighted average of each quality factor $q_j(c_i)$, ranging from **0** (*lowest* quality) to **1** (*highest* quality). FRANC ranks all code snippets based on their quality score $Q(c_i)$ and presents the best one to the developer. If multiple snippets have the same score, FRANC keeps the original order from the code generation model. For example, if the code snippets c_3 and c_8 have the same highest quality score, *i.e.*, $Q(c_3) = Q(c_8)$, then FRANC chooses c_3 to be in the first position.

E. Phase 5: LLM-based Code Quality Repair

Although FRANC has a quality-based ranking phase to help that the highest quality code snippet is given to the developer, it might be the case that the *top 1* code suggestion includes quality issues *because all generated suggestions had quality issues*. Consequently, that would require the engineer to go through the burden of *manually identifying* the quality problems in the generated code and *fixing* them. Hence, FRANC includes a *configurable automated repairing* phase that relies on the LLM itself to fix the problematic lines in the code snippet. The key insight of this phase is that code repair can be treated as a code generation task [44], [45], but with a prompt that explicitly instructs the model to fix a quality problem in a given location.

FRANC repairs code samples via *prompt engineering*. If the top-1 code snippet after the quality-based ranking has a quality score *below an acceptable threshold* τ (*i.e.*, $Q(c_1) < \tau$), then FRANC creates a **repair prompt** and re-pass this engineered prompt to the model to fix the code. This *repair prompt* is configurable, *i.e.*, it is up to the user to define the structure of this repair prompt. It is important to highlight that FRANC only attempts to repair the output only once.

For example, to repair the code snippet from Section III-C, one could configure FRANC to use the error message from Bandit to compose the repair prompt, which is then used again as a prompt to the model (CodeGen). This repair prompt concatenates the *previously generated code* (lines 1-8), the *found issues* (lines 9-10), and the *original prompt* (lines 11-16), as shown below:

```
1  import yaml
2  def yaml_load(filename):
3      '''
4      use the filename variable to open a file, load the
5      contents into a variable, and return it
6      '''
7      with open(filename) as f:
8          return yaml.load(f, Loader=yaml.FullLoader)
9  # FIX: At line no 8, Use of unsafe yaml load. Allows instantiation of arbitrary
10 # objects. Consider yaml.safe_load().
11 import yaml
12 def yaml_load(filename):
13     '''
14     use the filename variable to open a file, load the
15     contents into a variable, and return it
16     '''
```

IV. FRAMEWORK EVALUATION

To illustrate and evaluate the applicability and usefulness of our framework, we performed an empirical evaluation in which we implemented the components of FRANC outlined in Figure 1 (the Evaluation swimlane) to improve the quality of

Python and Java code automatically generated by five different LLMs. We focused on Python and Java because these are two of the most popular programming languages (based on a recent survey [46]). Moreover, we aimed to demonstrate how FRANC's configurable architecture enables a model-agnostic approach for improving the quality of generated code by choosing two languages and five different code generation models. In this evaluation, we answered the following research questions:

> **RQ1**: How well does the static filter correct and remove non-compilable code from the suggestion inventory?

Code generation models can output multiple suggestions for a single prompt, but not all suggestions are compilable [47]. Hence, our framework includes a *lightweight heuristic-based static filter* that automatically cleans the generated Python and Java code to remove any non-compilable code from the suggestion inventory (Phase 3). In this research question, we measure the effectiveness of this static filtering phase.

> **RQ2**: How well does the quality-based filter ranking system work?

In Phase 4, FRANC uses a quality-based ranking in order to sort eligible snippets (*i.e.*, code snippets that passed the filtering in Phase 3). Thus, we investigate whether FRANC's quality-based ranking performs better than the model's original ranking.

> **RQ3**: Can an LLM-based code generation model effectively repair code with quality issues?

FRANC includes a repairing phase (**5**) to fix a code snippet with quality issues using prompt engineering. It leverages the existing code generation model to fix the quality problem. Hence, we study the effectiveness of this LLM-based code-repairing approach.

> **RQ4**: How much overhead is introduced by the framework?

FRANC relies on static analyzers and is built on top of an existing code generation model. Therefore, it creates an extra overhead concerning the time to *filter* problematic snippets, *rank* the output from the code generation model, and *repair* it. This question explores how much overhead the framework introduced due to these additional phases.

In the next sections, we explain the methodology we followed to answer these four questions.

A. Prompts Creation

To answer our RQs, we retrieved prompts from existing code generation benchmarks [9], [18]–[21] that have been used by many prior works [47]–[52]. We also *created* our own set of prompts based on questions from StackOverflow. These prompts instruct the code generation models to generate a method/function's body based on the context in the docstring/-JavaDoc and the method/function's signature. We explain below each benchmark dataset used and our own dataset, which we refer to as **SOEVAL**.

- **AIXBENCH** [18] is a benchmark dataset that contains **175** prompts to evaluate the generation of Java code. The natural language description in the prompts is written in English and Chinese. We extracted **175** Java prompts with descriptions written in English.
- **CODEREVAL** [19] is a dataset containing 230 prompts for both Java and Python, retrieved from 43 and 10 open-source Python and Java projects hosted on GitHub, respectively. We retrieved a total of **460** prompts from this dataset.
- **MULTILINGUAL HUMANEVAL** [20] is a dataset with prompts for multiple programming languages created from the Python-based original **HumanEval** dataset [9]. We used **161** Java samples from Multilingual HumanEval and **164** from the original Python-based HumanEval dataset. Henceforth, we will refer to both datasets as simply **HUMANEVAL**.
- **SECURITYEVAL** is a benchmark dataset for evaluating Python code generation models from the perspective of security [21]. This dataset contains 130 prompts covering 75 entries from the Common Weakness Enumeration (CWE). However, we modified and removed prompts that explicitly instructed the model to generate vulnerable code. Thus, we have a total of **121** prompts covering 69 CWEs.
- **SOEVAL** is created by us by mining questions from Stack-Overflow. Our goal was to create a prompt dataset that reflects the real-life needs of software developers. To build this dataset, we first collected 500 popular and recent questions with Python and Java tags for each. From these 1,000 questions, we applied a set of inclusion and exclusion criteria. The inclusion criteria were: the question has to **(1)** explicitly ask *"how to do X"* in Python or Java; **(2)** include code in its body; **(3)** have an accepted answer that includes code. We excluded questions that were **(1)** open-ended and asking for best practices/guidelines for a specific problem in Python/Java; **(2)** related to finding a specific API/module for a given task; **(3)** related to errors due to environment configuration (*e.g.*, missing dependency library); **(4)** related to configuring libraries/API; **(5)** syntax-specific types of questions. By applying the criteria above to these 1K questions, we obtained **28** and **42** prompts for Java and Python, respectively.

Therefore, we had a total of **1,151** prompts, in which **594** and **557** prompts are for Java and Python, respectively.

B. Code Generation Models

We used the code generation models listed below (the first four are open-source, whereas the last one is closed-source) to create the *suggestion inventory*. The input of these models are the prompts previously collected (§ IV-A).

- **CodeParrot** [53] is a GPT-2 [54] model trained from scratch on Python code. It is fine-tuned on a clean and

deduplicated large dataset (>100GB). It can be used for code generation and other downstream tasks (*e.g.*, complexity prediction, code explanation, *etc.*). It has two versions: one is *CodeParrot-small* with 110 million parameters, and the other is the regular one with 1.5 billion parameters.

- **InCoder** [33] is a decoder-only transformer model [5] that can synthesize and edit code via infilling. It has one version with 1.3 billion parameters and another with 6.7 billion. We used the small version in our evaluation (1.3B).

- **CodeGen** [43] is a group of models for synthesizing programs using autoregressive languages. It has three types: *multi*, *mono*, and *nl*. The *multi* type is fine-tuned with multiple programming languages. The *mono* type is trained only with code written in Python. The *nl* type is fine-tuned mainly on natural language. Thus, we used the *multi* and *mono* model types. We used two versions for these models: one with 350 million parameters and the other with 2 billion.

- **PolyCoder** [49] is a family of three large open-source language models based on the GPT-2 architecture. They have been trained on a corpus of code from 12 different programming languages (*e.g.*, C/C++, C#, Go, Java, JavaScript, Python, Ruby, *etc.*), using about 25K repositories per language to build the corpus. The dataset was then pre-processed, deduplicated, and filtered, resulting in a training dataset size of 424GB. The parameters of these models range from 160 million to 2.7 billion parameters. We used all three versions of this model (160M, 400M, and 2.7B).

- **ChatGPT** is a closed-source chatbot developed by OpenAI [3]. It uses the GPT-3.5-turbo model (the latest version released on May 2023) and allows back-and-forth conversation with a user to generate code [47].

1) Suggestion Inventory Creation: We instructed each model to generate **ten** suggestions for each prompt. For **CodeParrot**, **InCoder**, **CodeGen**, and **PolyCoder**, we used the `transformers` library [55]. We instructed the model to generate additional 128 tokens after the prompt, *i.e.*, the model took the prompt and context as input and then generated the next probable token (*e.g.*, a keyword) that can come after this prompt and did not stop until 128 generated tokens. However, for **ChatGPT3.5**, we instructed it to generate 512 tokens. As it is optimized for conversations, it usually includes a textual explanation and the generated code with a modified version of the prompt. Hence, we extended the token size for this model and used the OpenAI API to generate results. Thus, we have **102,100** code samples in the suggestion inventory, *i.e.*, **41,010** Java code snippets, and **61,090** Python code snippets.

Token Limits Rationale: To decide the token size limits we ran a small experiment using **SECURITYEVAL** dataset. This dataset [21] includes an example of insecure code that can be generated from the given prompt. We tokenized these examples and found that they are 50 tokens in length, on average. Thus, we configured the open source models to generate 128 tokens ($\approx 2.5\times$ higher than the average number of tokens). For ChatGPT, as we explained before, we increased the limit to **512** to take into account explanations that are provided as part of the output (which consumes tokens).

C. Static Filtering

In our evaluation, we implemented six heuristics[1] to clean the generated code snippets before filtering them from the suggestion inventory based on their syntax check. We adopted the heuristics H_1, H_3, and H_6 from a recent study on unit test generation using LLMs [47]. The first heuristic (H_1) removes any text *before* and *after* backticks (*i.e.*, ``` `` ` code `` ` ```). The second heuristic (H_2) adds *part* of the prompt or the *full* prompt if it is not found in the generated code (otherwise, the code fails the syntax check because it does not include the function/method signature, import statements, *etc.*). Heuristics H_3 and H_4 are Python-specific and remove extra code after the target method. The third heuristic (H_3) removes any code found *after* a `"\n``` `\n\n##"`, or `"\n</code>"` pattern (*i.e.*, it removes extra code). The fourth Python-specific heuristic (H_4) removes any additional code *after* the target method-/function. The fifth heuristic (H_5) removes any code in an extra Java class (*i.e.*, it only keeps the Java class mentioned in the prompt). The sixth heuristic (H_6) fixes incomplete code by iteratively deleting lines (from bottom to top) and adding 1-2 curly brackets for a Java code. Moreover, if the prompt was meant to repair quality problems (§ IV-E), we apply an additional heuristic (H_7) before applying these six heuristics. This heuristic replaces the old problematic code (*i.e.*, containing a code/security smell) with the newly repaired source code.

D. Quality-based Ranking

Up to this point, the generated suggestions are compilable but may still have quality issues, such as code and security smells [10]. We used Bandit to discover security smells in Python code and SpotBugs to discover code smells in Java code. In our evaluation, we configured FRANC to employ the quality score defined in Equation 2. By using this quality scheme, we can move up in the rank of the 10 generated suggestions for a prompt that is free of code/security smells.

$$Q(c_i) = \begin{cases} 1, & \text{if } c_i \text{ is compilable and free of smells} \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

E. Repair by Prompt Engineering

Recall that if the first suggestion is below a quality threshold ($Q(c_1) < \tau$), then FRANC attempts to repair c_1 through prompt engineering. In our evaluation, we set the threshold τ to one, which means that FRANC will repair the first suggestion if it has *at least one smell* in it (*i.e.*, $Q(c_1) < 1$). We configured FRANC to use the error metadata provided by Bandit and SpotBugs to repair the generated code for the top-1 suggestion ranked by FRANC in the previous phase. Specifically, we used the code smell's description and location to craft a repair prompt using three different structures (**P1**,

---

[1]Due to space constraints, we superficially describe the heuristics in here, but we detail each of them with examples in our supplementary materials.

		CodeParrot (small)	CodeParrot (regular)	InCoder (1B)	CodeGen (350M-mono)	CodeGen (350M-multi)	CodeGen (2B-mono)	CodeGen (2B-multi)	PolyCoder (160M)	PolyCoder (0.4B)	PolyCoder (2.7B)	GPT3.5
Java	Before	-	-	0.15%	-	5.86%	-	7.43%	0.13%	0.29%	0.19%	0.89%
	After	-	-	8.72%	-	21.65%	-	31.77%	11.30%	13.08%	14.66%	46.91%
	% Increase	-	-	8.57%	-	15.79%	-	24.33%	11.17%	12.79%	14.47%	46.02%
Python	Before	22.22%	25.98%	55.91%	34.56%	37.06%	34.29%	40.49%	19.73%	21.33%	22.72%	57.75%
	After	63.69%	68.96%	68.70%	68.39%	71.41%	73.28%	71.83%	59.25%	62.40%	65.02%	68.52%
	% Increase	41.47%	42.99%	12.79%	33.84%	34.35%	38.99%	31.34%	39.52%	41.07%	42.30%	10.77%

P2, and **P3**) and send it back to the code generation model to generate 10 suggestions again. To illustrate these different prompt repair structures, consider the generated code to be repaired shown in Listing 3 (where the highlighted lines were the original prompt used to generate this code). The prompt structure **P1** adds source code comments *after* the code to be repaired in the format "`Fix: At line « # »`, `« error msg »\n Fixed Code:\n`", as shown in Listing 3 (lines 10-11). The error message comes from Bandit (for Python snippets) and Spotbugs (for Java snippets).

```
                    ─── Code to be repaired ───
1  from django.db import connection
2  def show_user(request, username):
3      '''
4      Create a cursor to get user info from users table and return it
5      '''
6      cursor = connection.cursor()
7      cursor.execute("SELECT * FROM users WHERE username = '%s'" % username)
8      user = cursor.fetchone()
9      return user
            ─── Added comments to be used as repair prompt ───
10 # Fix: At line 7, Possible SQL Injection
11 # Fixed Code:
```

Listing 3: Generated code with a SQL injection (line 7)

The second prompt structure (**P2**) appends to the code to be repaired a *comment* followed by the *original prompt*. For example, the repair prompt for the code in Listing 3 would include the lines 1–11 followed by lines 1-5 (*i.e.*, the original prompt). The third prompt repair structure (**P3**) only includes the *code to be repaired* up to the first line that has an issue, followed by the *fix message*. For instance, while repairing the code in Listing 3, this prompt would only include lines 1–6 followed by lines 10–11.

It is important to clarify that the same code snippet can have *multiple* issues. Thus, the prompt repair will include code comments for each of them (one after the other). Moreover, when SpotBugs produces messages without any specific line number, the repair prompt only includes the message but not the line (*i.e.*, `// Fix: <Spotbugs Message>`). Lastly, we ignored cases in which the error is located in the original prompt (*e.g.*, importing an unused class).

In the end, we have **1,023** repair prompts (341 for each prompt repair structure). We then regenerate 10 suggestions for each of these repair prompts using the same model generated by the code under repair.

V. EVALUATION RESULTS

A. RQ1: Static Filter Effectiveness

Recall that FRANC applies a heuristic-based static filter to clean the generated code and filter out uncompilable snippets in Phase **3**. Table I shows the percentage of code snippets that are compilable *before* and *after* FRANC applies its static filter. We can see that the improvements in the *percentage of compilable suggestions* range from **8.6%** to **24.3%** for Java with open-source models, and **46%** for Java with ChatGPT-3.5. Our static filter also improves the *number of prompts with at least one compilable suggestion*. In the end, **22%**, **69%**, **73%**, **55%**, **54%**, **51%**, **61%** of the prompts used for InCoder-1B, CodeGen-350M-multi, CodeGen-2B-multi, PolyCoder-160M, PolyCoder-0.4B, PolyCoder-2.7B, and GPT-3.5, respectively, had *at least one compilable Java suggestion* after FRANC applied its filter (the increases ranged from **20%** to **59%**).

For Python, the improvement in the number of compilable suggestions ranges from **12.8%** to **43%** for open-source models, and for GPT-3.5, it is **10.8%**. It also improves the number of prompts with at least one suggestion (for open-source models, the improvements range from **0.80%** to **10.27%**, and for GPT-3.5, it is **6.43%**). These improvements are smaller when compared to Java because the models already produce **85.4%** prompts with at least one compilable Python suggestion, on average. In contrast, the average percentage of prompts with at least one compilable suggestion is only **1.7%** for Java.

> **RQ1 Findings:** FRANC's static filtering phase can increase the compilation rates of the code generated by the studied models. The improvements were more noticeable for Java code, where less than 2% of prompts had at least one compilable Java snippet, on average.

B. RQ2: Quality-based Ranking Effectiveness

In RQ2, we investigate FRANC's effectiveness in *ranking* code snippets. To do so, we calculated the Normalized Discounted Cumulative Gain at k (NDCG@k) [56]. The NDCG@k measures how well k results are sorted as follows [57]:

$$NDCG_{@k} = \frac{\sum_{i=1}^{k} \frac{rel_i}{log_2(i+1)}}{IDCG_{@k}} \qquad (3)$$

The term rel_i in this equation indicates the relevance score of c_i (*i.e.*, the code snippet c at the position i). It ranges from 0 to 3, where 0 means the *lowest* relevant suggestion and 3 indicates the *highest* relevant suggestion. If the code snippet c_i is not compilable, the score is **0**. If c_i is compilable, but has a quality issue (*i.e.*, $Q(c_i) = 0$), then its relevance score

TABLE II
NDCG@10 Scores for the Original Model Ranking and Franc Ranking

		CodeParrot (small)	CodeParrot (regular)	InCoder (1B)	CodeGen (350M-mono)	CodeGen (350M-multi)	CodeGen (2B-mono)	CodeGen (2B-multi)	PolyCoder (160M)	PolyCoder (0.4B)	PolyCoder (2.7B)	GPT3.5
	# Prompts			4	28			37	15	17	17	37
Java	Model's # prompts ($rel_1 = 3$)	-	-	0	3	-	-	8	1	1	1	18
	FRANC's # prompts ($rel_1 = 3$)	-	-	0	7	-	-	24	2	4	1	30
	Model's NDCG@k	-	-	0.0979	0.2330	-	-	0.3019	0.1385	0.1525	0.2264	0.5775
	FRANC's NDCG@k	-	-	0.1319	0.3189	-	-	0.4143	0.2312	0.2635	0.3174	0.6695
	# Prompts	13	14	3	17	18	21	19	17	18	20	11
Python	Model's # prompts ($rel_1 = 3$)	1	0	1	0	2	6	4	1	2	2	1
	FRANC's # prompts ($rel_1 = 3$)	1	0	2	5	3	8	6	1	4	3	6
	Model's NDCG@10	0.3297	0.4021	0.2012	0.3944	0.3630	0.4745	0.4738	0.3740	0.4200	0.4541	0.3742
	FRANC's NDCG@10	0.3853	0.4628	0.2419	0.4581	0.4671	0.5453	0.5335	0.4429	0.4941	0.5209	0.4630

is **1**. If c_i is compilable and free of quality problems (*i.e.*, $Q(c_i) = 1$) but does not fully implement the prompt's intent (*i.e.*, it is functionally incorrect), the score is **2**. A code snippet c_i that is compilable, does not have a quality problem, and is functionally correct has a relevance score equal to **3**. Given that we generated 10 suggestions per prompt, the value of k is equal to 10. The $IDCG_{@k}$ in Eq. 3 is the *ideal discounted cumulative gain*, which is a normalization factor used to ensure the output ranges from 0 to 1. It is equal to the highest possible value achieved when all results are correct, *i.e.*, $IDCG_{@10} = \frac{3}{log_2(1+1)} + ... + \frac{3}{log_2(10+1)} \approx 13.631$.

In total, we have **2,271** prompts with *at least one quality issue* from different models and dataset combinations that need to be repaired. Since it would be time-consuming to manually analyze 22,710 code snippets (*i.e.*, 2,271 prompts $\times$ 10 suggestions), we instead randomly chose a subset of **326** prompts (95% confidence level) to manually analyze. The number of prompts per model kept the same proportions as the original set of **2,271** prompts.

The relevance scores 0 and 1 are automatically computed based on the syntax check and quality score computed in Phases 3 and 4, respectively. For the remaining snippets (*i.e.*, that are compilable and free of smells), the relevance score is assigned manually by two researchers. They independently provide the rating by judging the intention of the prompt and their reflection on the generated suggestion. The Cohen's kappa score of the inter-raters' agreement is 0.815, which indicates a strong agreement between the raters [58]. We resolved the disagreements through discussion. After resolving these disagreements, we computed the $NDCG_{@10}$ for FRANC and compared it with the $NDCG_{@10}$ for the original rank produced by the underlying model.

Table II shows the average $NDCG_{@10}$ for each model before using FRANC (*i.e.*, the original rank) and after using it. FRANC increases the $NDCG_{@10}$ for all models and languages. The highest improvement for Java was from **0.3019** to **0.4143** (CodeGen 2B-multi). For Python, the highest increase was from **0.3630** to **0.4671** (CodeGen 350M-mono). We observe a similar improvement trend for both languages. The average $NDCG_{@10}$ difference between FRANC and the original models' ranking output is **0.096** for Java and **0.07** for Python. A

paired t-test comparing the $NDCG_{@10}$ shows a statistically significant difference for both languages (p<0.0001).

Table II also shows how many prompts we manually analyzed per model and how many prompts in which $rel_1 = 3$ (*i.e.*, c_1 is compilable, functionally correct, and smell-free) before and after using FRANC. We find that the number of prompts with $rel_1 = 3$ also increases after using FRANC.

RQ2 Findings: FRANC's quality-based ranking increases the $NDCG_{@10}$ (statistically significant difference) for both Python and Java. FRANC also increases the number of prompts in which the first suggestion is compilable, functionally correct, and free of smells.
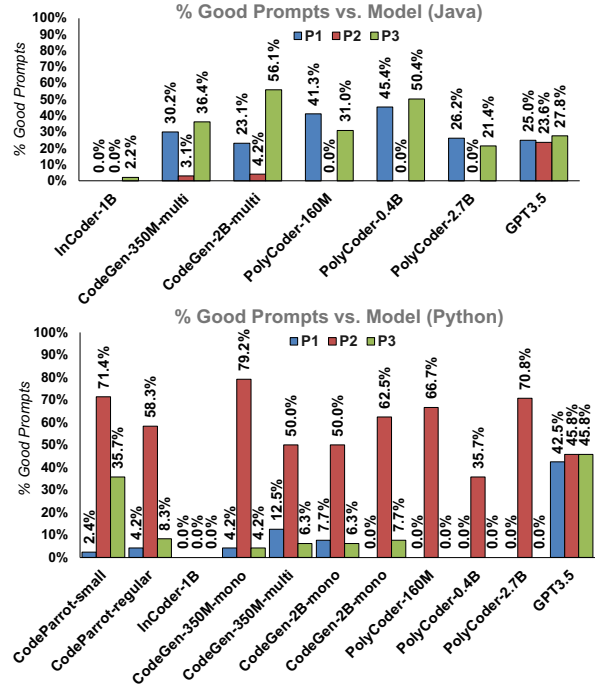

Fig. 2. Prompt Engineering-based Repair for Java and Python Benchmarks

C. RQ3: Code Repair Effectiveness

After ranking the suggestions based on quality (Phase 4), if the re-sorted top-1 suggestion (*i.e.*, c_1) still has a quality issue (*i.e.*, $Q(c_i) = 0$), then FRANC repairs the problematic top-1

TABLE III
TIME TO RUN EACH PHASE IN SECONDS

		CodeParrot (small)	CodeParrot (regular)	InCoder (1B)	CodeGen (350M-mono)	CodeGen (350M-multi)	CodeGen (2B-mono)	CodeGen (2B-multi)	PolyCoder (160M)	PolyCoder (0.4B)	PolyCoder (2.7B)	GPT3.5
Java	Filtering Phase	-	-	0.022837	-	0.016746	-	0.014988	0.014052	0.015473	0.014412	0.034090
	Ranking Phase	-	-	1.708652	-	2.020211	-	2.195032	2.098262	1.804040	1.845289	2.087183
	Repairing Phase	-	-	0.000023	-	0.000038	-	0.000037	0.000037	0.000040	0.000030	0.000045
	Total	-	-	1.731512	-	2.036994	-	2.210057	2.112351	1.819553	1.859731	2.121319
Python	Filtering Phase	0.000053	0.000053	0.000055	0.000048	0.000048	0.000049	0.000048	0.000047	0.000047	0.000046	0.002357
	Ranking Phase	0.085460	0.085794	0.062640	0.088703	0.084753	0.092222	0.085266	0.083774	0.083833	0.088145	0.087277
	Repairing Phase	0.000098	0.000022	0.000011	0.000019	0.000015	0.000016	0.000018	0.000014	0.000027	0.000015	0.000021
	Total	0.085610	0.085869	0.062705	0.088770	0.084816	0.092286	0.085331	0.083834	0.083908	0.088206	0.089655

suggestion through prompt engineering (*i.e.*, it creates a repair prompt and sends it back to the model). Recall that we studied three different prompt repair structures (P1–P3). Figure 2 depicts the percentage of prompts with at least one repaired suggestion per model. Prompts with at least one snippet c_i in which $Q(c_i) = 1$ are referred to as *"good prompts"*.

The prompt structure **P2** is not performing well in repairing Java code. It is not generating any good prompts for InCoder and PolyCoder models. For Java, **P3** is the best-performing prompt repair structure. It can generate good prompts from 2.2% to 56.1% of the time. However, we see a different scenario for Python in Figure 2. For Python, **P2** is performing better except for GPT-3.5, though the difference between **P2** and **P1** is very close. The prompt structure **P2** can produce good prompts from 35.71% to 79.2%. We also observe that, overall, the models perform better in repairing Python samples than Java samples. One possible explanation for this observation is that these LLMs are heavily trained/fine-tuned with Python samples [9].

> **RQ3 Findings:** FRANC can effectively repair a top-1 suggestion that does not have the highest quality. Different programming languages may need different prompting engineering structures, but FRANC can produce up to 80% prompts with at least one good suggestion.

D. RQ4: Extra Overhead from the Framework

Table III summarizes the extra overhead taken by FRANC. To ensure consistency of measurements, we used the same machine while running the experiment (an Apple M1 Chip with 8 GB RAM). We can see that running all the phases for Java; takes **1.984502** seconds on average (standard deviation of $\sigma = 0.06$). For Python, it takes **0.084635** seconds, on average, ($\sigma = 0.003$). The *ranking phase* was the most time-consuming phase, as it needed to run external tools to compute the quality score for each code snippet c_i. Although FRANC adds an overhead to filter, rank and repair code snippets, it does not require any fine-tuning of the models, which would be time-consuming, costly, and resource-hungry.

> **RQ4 Findings:** Although FRANC adds an overhead, the extra time needed is less than 2 seconds (on average). The ranking phase is FRANC's most time-consuming phase.

VI. DISCUSSION

A. Infilling vs Synthesis vs Chat-based Generation

Our work investigates five models with six prompt datasets crafted from different sources (*e.g.*, programming problems [9], StackOverflow, *etc.*). CodeParrot [53], PolyCoder [49], and CodeGen [43] focuses on **program synthesis**, *i.e.*, they take a prompt as input to generate code *after* the prompt. InCoder [33] models focuses on **infilling**; it fills up with code between a prompt by taking from *both* sides. GPT-3.5 [3] focuses on **conversation-style** code generation. Although FRANC is geared towards program synthesis, it still improves the performance of infilling models like InCoder [33]. For example, our results showed that without a static filter, no compilable suggestions were produced by InCoder (§ V-A). FRANC was able to clean up InCoder's output using heuristics such that 22% of prompts had at least one compilable snippet. Though the result of this infilling model is not as great as other models (*i.e.*, lower $NDCG_{@10}$ improvement), FRANC helped to show code with higher quality in the first position.

GPT-3.5 [3] is optimized for multi-turn style conversation with human feedback. It performs better than most of the models for generating suggestions, and FRANC significantly improves the ranking of the suggestions. This model better understands different repair scenarios, whereas open-source models respond to different scenarios depending on the programming language.

B. Code Repairing using LLM

We used prompt engineering techniques to repair code and security smells using LLMs. We found that for Python, LLMs are better able to solve issues related to XML validation vulnerabilities (*i.e.*, CWE-20: *Improper Input Validation*), using APIs from `subprocess` library (*i.e.*, CWE-78: *OS Command Injection*) and a Flask application with `debug=`**`True`** (*i.e.*, CWE-94: *Code Injection*). Conversely, LLMs were less capable of solving issues related to the *Use of a Broken or Risky Cryptographic Algorithm* (CWE-327), *Path Traversal* (CWE-22), and *Incorrect Permission Assignment for Critical Resource* (CWE-732).

For Java, LLMs can resolve code smells related to the *invocation of toString on an array*, *suspicious reference comparison*, and *return value of method without side effect is ignored*. How-

ever, LLMs can hardly repair *infinite loops*, *array indexing out of bounds*, and *useless control flow to next line*.

VII. Threats to Validity

A threat to the *external validity* of this work is that we only investigated transformed-based [5] LLMs. However, current commercial products are built based on these types of models [2], [3]. Another external validity threat is that we used default hyperparameters values with 128 tokens to generate source code for the open-source LLMs and 512 tokens for ChatGPT. Thus, we acknowledge that our results may not generalize for other inference hyperparameters. However, our work established the importance of a framework, FRANC, and showed it was independent of the code generation models. We also used two external tools (Bandit [59], and Spotbugs [60]) in the framework. Practitioners and researchers widely use these tools [10], [61].

A threat to the *internal validity* of this work is that we manually analyzed ranked code snippets to compute their relevance score (§ V-C), which is prone to biases. However, to ensure that biases are removed, we conducted a peer review of these analyses and reported our Cohen's kappa score [58] (which showed strong agreement). Another *internal validity* threat is that we manually curated a new prompt dataset called SOEVAL. However, since StackOverflow is a popular Q&A website used by developers, this dataset can be a proxy for real-world developers' prompts.

VIII. Related Work

Program synthesis refers to automatically generating a program that satisfies the user's intent given as high-level specifications or input-output examples [62]. One of the foundations of program synthesis is deductive synthesis [63], [64], where a task specification is transformed into constraints, and the program is extracted after demonstrating its ability to satisfy the constraints [62]. An example of this approach comes from Yin *et al.* [65]. This work mapped text to abstract syntax trees using recurrent networks. These abstract syntax trees were then coded using attention.

Many LLMs have been produced with the goal of generating code, such as CodeBert [30], Codex [9], and CodeT5 [32]. Another form of a code generation model that produces code for competitive programming challenges is AlphaCode [66]. GitHub Copilot [2], a closed-source tool for code generation, uses the upgraded version of Codex to develop an improved auto-complete mechanism. Additionally, other recent works [6]–[8] focused on optimizing the process to create, fine-tune, and infer the LLM-based code generation technique. However, our work focuses on being a *lightweight* approach to filter, rank, and repair code snippets in an existing model's output from a *code quality* perspective.

After GitHub Copilot's commercial release to users [2], prior works [9], [32], [66] expressed their concern about security, privacy, and bias in the generated code. A recent study found that GitHub Copilot can produce unsafe (vulnerable) code [11]. Another study by Siddiq *et al.* [10] observed the presence of code smells (including security smells) in code generation training sets and their leakage to the models' output. Another recent work [12] conducted a user study to investigate the security implication of GitHub Copilot as a large language model-based code assistant. Rather than performing an empirical study on the quality or security issues in the code generation paradigm, our work introduces a novel framework to address quality problems by reducing how often vulnerable and low-quality code is exposed to developers from code generation models.

Other works have aimed to improve these code generation models' output by directly changing them. He and Vechev [67] used property-specific continuous vectors to guide program generation toward the given property without modifying the LLM's weights; specifically, they tried to generate secure source code without compromising the program's correctness. However, this prior work is limited to certain CWEs and a code generation model that needs fine-tuning. In contrast, our work studies how to resolve quality issues without re-training or fine-tuning an LLM and is not limited to a certain quality attribute; developers can configure FRANC's quality score (§ III-D) in any way they wish to give more weight to certain quality attributes over the others.

Chen *et al.* [68] modified these code generation models by introducing human feedback in natural language to the language model during training by using the language model's natural ability to incorporate feedback. When the model generates a code snippet, it will use feedback from the user to repair the snippet if it is faulty. Our work differs from these works because rather than changing the model itself, which would be costly, we use the original model and filter out vulnerable and low-quality code from the possible responses the model could produce.

IX. Conclusion and Future Work

Although automated code generation tools can help developers to speed up software development, the generated source codes must also be maintainable, high quality, and free of code smells. As automated generated codes are mixed with human-written code, it is necessary to ensure the quality of the generated code so that it does not introduce reliability issues. Our framework, FRANC, helps get vulnerability-free output and comparatively high-quality source code. Our work introduced the first-of-its-kind lightweight framework for improving the quality of the generated code with practical usage of code repairing using LLMs to remove quality issues. We demonstrated how our framework performs by using it to improve the quality of the Java/Python code generated by five LLMs. In the future, we will evaluate FRANC using other programming languages and learning models.

REFERENCES

[1] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Comput. Surv.*, vol. 53, no. 3, jun 2020.

[2] G. Inc., "Github copilot : Your ai pair programmer," 2022, [Online; accessed 10. Oct. 2022]. [Online]. Available: https://copilot.github.com

[3] "Chat completions," Accessed Mar 25, 2023, 2023. [Online]. Available: https://platform.openai.com/docs/guides/chat

[4] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proc. of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming*, ser. MAPS 2022.   New York, NY, USA: ACM, 2022, p. 21–29.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30.   Curran Associates, Inc., 2017.

[6] A. Madaan, S. Zhou, U. Alon, Y. Yang, and G. Neubig, "Language models of code are few-shot commonsense learners," 2022. [Online]. Available: https://arxiv.org/abs/2210.07128

[7] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[8] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35.   Curran Associates, Inc., 2022, pp. 21314–21328.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto *et al.*, "Evaluating large language models trained on code," 2021.

[10] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *2022 IEEE 22nd Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, 2022.

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

[12] G. Sandoval, H. Pearce, T. Nys, R. Karri, B. Dolan-Gavitt, and S. Garg, "Security implications of large language model code assistants: A user study," *arXiv preprint arXiv:2208.09727*, 2022.

[13] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 1, Feb 2015.

[14] T. Sharma, M. Fragkoulis, and D. Spinellis, "House of cards: code smells in open-source c# repositories," in *2017 ACM/IEEE Intl. Symposium on Empirical Software Engineering and Measurement (ESEM)*.   IEEE, 2017, pp. 424–429.

[15] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[16] K. W. Church, Z. Chen, and Y. Ma, "Emerging trends: A gentle introduction to fine-tuning," *Natural Language Engineering*, vol. 27, no. 6, p. 763–778, 2021.

[17] "Openai pricing," Accessed May 5, 2023, 2023. [Online]. Available: https://openai.com/pricing

[18] Y. Hao, G. Li, Y. Liu, X. Miao, H. Zong, S. Jiang, Y. Liu, and H. Wei, "Aixbench: A code generation benchmark dataset," 2022.

[19] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, T. Xie, and Q. Wang, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," 2023.

[20] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang, "Multi-lingual evaluation of code generation models," in *The Eleventh Intl. Conf. on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=Bo7eeXm6An8

[21] M. L. Siddiq and J. C. S. Santos, "Securityeval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques," in *Proc. of the 1st Intl. Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S22)*, 2022.

[22] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[23] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, mar 2020.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*.   Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[25] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[26] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33.   Curran Associates, Inc., 2020, pp. 1877–1901.

[27] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *44th Intl. Conf. on Software Engineering (ICSE)*, 2022.

[28] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd Intl. Conf. on Software Engineering (ICSE)*.   IEEE, 2021, pp. 150–162.

[29] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th Intl. Conf. on Mining Software Repositories (MSR)*.   IEEE, 2021, pp. 329–340.

[30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*.   Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[31] Y. Gao and C. Lyu, "M2ts: Multi-scale multi-modal approach based on transformer for source code summarization," in *Proc. of the 30th IEEE/ACM Intl. Conf. on Program Comprehension*, ser. ICPC '22.   New York, NY, USA: Association for Computing Machinery, 2022, p. 24–35.

[32] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing*.   Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708.

[33] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *CoRR*, vol. abs/2204.05999, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.05999

[34] S. H. Kan, *Metrics and models in software quality engineering*. Addison-Wesley Professional, 2003.

[35] N. C. Guido van Rossum, Barry Warsaw, "Pep 8 — the style guide for python code," 2022, [Online; accessed 30. Sept. 2022]. [Online]. Available: https://pep8.org/

[36] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[37] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code Smells Detection and Visualization: A Systematic Literature Review," *Archives of Computational Methods in Engineering*, vol. 29, no. 1, pp. 47–94, Jan. 2022.

[38] A. Gupta, B. Suri, V. Kumar, S. Misra, T. Blažauskas, and R. Damaševičius, "Software code smell prediction model using shannon, rényi and tsallis entropies," *Entropy*, vol. 20, no. 5, p. 372, 2018.

[39] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," in *2019 IEEE/ACM 41st Intl. Conf. on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 164–175.

[40] M. R. Rahman, A. Rahman, and L. Williams, "Share, but be aware: Security smells in python gists," in *2019 IEEE Intl. Conf. on Software Maintenance and Evolution (ICSME)*, 2019, pp. 536–540.

[41] M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," in *2017 IEEE 17th Intl. working Conf. on source code analysis and manipulation (SCAM)*. IEEE, 2017, pp. 121–130.

[42] G. Inc., "Use of a broken or weak cryptographic hashing algorithm on sensitive data," 2022, [Online; accessed 30. Oct. 2022]. [Online]. Available: https://codeql.github.com/codeql-query-help/python/py-weak-sensitive-data-hashing/

[43] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint*, 2022.

[44] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1–18.

[45] H. Joshi, J. Cambronero, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen, "Repair is nearly generation: Multilingual program repair with llms," 2022. [Online]. Available: https://arxiv.org/abs/2208.11640

[46] S. Cass, "Top Programming Languages 2022," *IEEE Spectrum*, Aug. 2022. [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2022

[47] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Exploring the effectiveness of large language models in generating unit tests," 2023.

[48] J. P. Inala, C. Wang, M. Yang, A. Codas, M. Encarnación, S. Lahiri, M. Musuvathi, and J. Gao, "Fault-aware neural code rankers," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 13 419–13 432.

[49] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. of the 6th ACM SIGPLAN Intl. Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–10.

[50] Y. Dong, J. Ding, X. Jiang, Z. Li, G. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," 2023.

[51] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," 2023.

[52] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "Skcoder: A sketch-based approach for automatic code generation," 2023.

[53] L. von Werra, "Codeparrot," 2022. [Online]. Available: https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot

[54] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[55] H. Inc., "Huggingface transformers," 2022, [Online; accessed 20. Oct. 2022]. [Online]. Available: https://huggingface.co/docs/transformers/

[56] K. Järvelin and J. Kekäläinen, "Ir evaluation methods for retrieving highly relevant documents," in *Proc. of the 23rd Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, ser. SIGIR '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 41–48.

[57] ——, "Cumulated gain-based evaluation of ir techniques," *ACM Trans. Inf. Syst.*, vol. 20, no. 4, p. 422–446, oct 2002.

[58] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[59] B. Developers, "Bandit," 2022, [Online; accessed 28. June. 2022]. [Online]. Available: https://bandit.readthedocs.io/

[60] "Spotbugs: Find bugs in java programs," Accessed Mar 25, 2023, 2023. [Online]. Available: https://spotbugs.github.io

[61] D. A. Tomassi, "Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs," in *Proc. of the 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, 2018, pp. 980–982.

[62] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[63] C. Green, "Application of theorem proving to problem solving," in *Proc. of the 1st Intl. Joint Conf. on Artificial Intelligence*, ser. IJCAI'69. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1969, p. 219–239.

[64] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Commun. ACM*, vol. 14, no. 3, p. 151–165, mar 1971.

[65] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 440–450.

[66] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[67] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," 2023.

[68] A. Chen, J. Scheurer, T. Korbak, J. A. Campos, J. S. Chan, S. R. Bowman, K. Cho, and E. Perez, "Improving code generation by training with natural language feedback," 2023.

12