2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest) | 979-8-3315-0190-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/DeepTest66595.2025.00011

Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation

Benjamin Steenhoek*, Michele Tufano†, Neel Sundaresan*, Alexey Svyatkovskiy‡
*Microsoft Data & AI, {bensteenhoek,neels}@microsoft.com
†Google, ‡Google DeepMind, {tufanomichele,alexeysv}@google.com

***Abstract*—Software testing is a crucial but time-consuming aspect of software development, and recently, Large Language Models (LLMs) have gained popularity for automated test case generation. However, because LLMs are trained on vast amounts of open-source code, they often generate test cases that do not adhere to best practices and may even contain test smells (anti-patterns). To address this issue, we propose Reinforcement Learning from Static Quality Metrics (RLSQM), wherein we utilize Reinforcement Learning to generate high-quality unit tests based on static analysis-based quality metrics. First, we analyzed LLM-generated tests and show that LLMs frequently do generate undesirable test smells — up to 37% of the time. Then, we implemented lightweight static analysis-based reward model and trained LLMs using this reward model to optimize for five code quality metrics. Our experimental results demonstrate that the RL-optimized Codex model consistently generated higher-quality test cases than the base LLM, improving quality metrics by up to 23%, and generated nearly 100% syntactically-correct code. RLSQM also outperformed GPT-4 on all code quality metrics, in spite of training a substantially cheaper Codex model. We provide insights into how reliably utilize RL to improve test generation quality and show that RLSQM is a significant step towards enhancing the overall efficiency and reliability of automated software testing. Our data are available at this link: https://doi.org/10.6084/m9.figshare.25983166.**

I. Introduction

Software testing is a crucial component in the development of reliable and robust software systems. In addition, tests must be maintainable to promote efficient software development and bug detection. *Test smells* encompass a range of issues that can hinder the comprehensibility, maintainability, and overall quality of a test suite. Numerous studies have highlighted the detrimental impact of test smells on various aspects of software testing [48, 6, 46, 41, 30, 42]. In particular, Bavota et al. [6] showed that test smells are correlated with decreased comprehensibility and maintainability of the test code, and Spadini et al. [41] demonstrated a connection between test smells and increased susceptibility to changes and defects in the production code. Consequently, test smells must be addressed to ensure the effectiveness of software testing efforts.

Large Language Models (LLMs) have become popular for automatically generating code [10, 16], including test cases [19, 49, 51, 39, 12]. However, because LLMs are trained on open source code, they often incorporate undesirable properties seen in real-world test suites. In code completion, this manifests when LLMs generate vulnerable code [29] or leak sensitive data [9, 23]. We show in Section IV-B that pre-trained models often exhibit test smells when generating tests.

Recently, Reinforcement Learning (RL) has been used to train LLMs to follow instructions [26], instill specific qualities [5, 4, 45] and generate higher-quality code [36, 17]. RL addresses *exposure bias* [7, 33] which arises during supervised fine-tuning (SFT), where models are trained on ground-truth text but rely on self-generated outputs to generate the next word, causing a distribution mismatch. It also avoids the additional inference cost of prompting approaches like in-context learning [20] and tree-of-thoughts [50].

To address these issues, we introduce *Reinforcement Learning from Static Quality Metrics (RLSQM)*, which uses RL to align LLMs with a static analysis-based reward model. We leverage the framework of Reinforcement Learning from Human Feedback (RLHF) [24, 52, 26, 43, 11], which overcomes the data limitations of traditional fine-tuning by training LLMs to align with a reward model. Instead of relying on expensive, unpredictable, and often biased human feedback, we use fully automated static analysis to detect well-known quality metrics. We train a Reward Model (RM) to score test cases based on these quality metrics, then use it to provide feedback for the Proximal Policy Optimization (PPO) [37] algorithm to train LLMs to generate test cases that maximize the expected reward (i.e., higher quality tests).

We begin by generating test cases using base LLMs and investigating their alignment with testing best practices and their susceptibility to test smells (§ III-A). Our results show that LLM-generated test cases often fail to follow best practices contain undesirable test smells; specifically, base Codex generated 17% tests with incorrect syntax, 31% tests without assertions, and 37% tests without calls to the focal method (method under test).

Next, we apply RLSQM and show that it enables LLMs to generate substantially higher-quality tests (§ III-B). We explored several ablations and strategies to optimize models for multiple quality metrics, providing valuable insight into effective RL fine-tuning for test generation. RL-finetuned Codex models generated more tests with best practices and fewer containing test smells: up to 23.2% more tests with Assertions and 17.9% more tests with Focal calls, and up to 2.22% fewer tests with Duplicate Assertions and 2.53% fewer tests with Conditionals/Exceptions. Furthermore, compared to the substantially more expensive GPT-4 model, RL-finetuned models yielded greater performance on five out of seven quality metrics, including all code-quality metrics, at a substantially lower inference cost.

979-8-3315-0190-7/25/$31.00 ©2025 IEEE
DOI 10.1109/DeepTest66595.2025.00011

In summary, we make several contributions:

- We analyze over 6 million LLM-generated tests with respect to Best Practices, Documentation, and Test Smells.
- We introduce Reinforcement Learning from Static Quality Metrics (RLSQM), a fully-automatic approach for fine-tuning language models to generate high-quality tests.
- We train and evaluate RLSQM on a dataset of 62 thousand C# focal methods from 82 open-source projects.
- We explore diverse strategies for training with RLSQM, including Individual, Sequential, and Combined Rewards, and report on settings which were critical for convergence.

II. Approach

In RLSQM, we follow these steps: we collect open-source methods we want to test (§ II-A), generate unit tests with base LLMs (§ II-B), automatically evaluate the quality of the tests with Quality Analyzer (§ II-C), filter to a set of high-quality "golden" tests for supervised fine-tuning (§ II-D), then use Quality Analyzer directly in RL fine-tuning (§ II-E).

A. Data collection

TABLE I: Statistics of the dataset of focal methods.

Statistic	Value
# projects	82
# focal classes	13,567
# focal methods	62,103
Mean # SLOC per focal method	6.5
Mean # parameters per focal method	1.6

We constructed our dataset by curating focal methods from open-source C# projects on GitHub. Initially, we identified 100 non-fork projects predominantly written in C# with a minimum of 5 stars. Among these, we exclusively considered projects featuring TestMethods written in the MSTest framework [21], ultimately selecting 82 repositories. From these repositories, we extracted all public methods for which the developer wrote tests, in order to focus our study on the methods which developers specifically intended to assess. Our dataset consists of 62k focal methods sourced from these 82 projects, averaging around 757 focal methods per project. Table I lists the statistics characterizing our dataset.

B. Generating Unit Tests with Language Models

We formulate the unit test generation task as completing a test case based on a focal method and its context – the focal class source code and relevant file-level context. This formulation aligns with the causal language modeling objective employed during pre-training. While we employed this prompt design because it consistently performed the best in our pilot studies, we emphasize that our findings are orthogonal to prompt design, and can be integrated with a diverse set of prompts.

Figure 1 details the prompt's contents (top to bottom): (1) The filepath containing the focal method; (2) The focal method's source code and accompanying context; (3) A "hint" specifying a hypothetical filepath for the test code and the beginning of the test method signature, which conditions the model to test the focal method.

```
src/Commands/BenchmarkCommand.cs:
using System;
using System.Linq;
using System.Threading.Tasks;

public class BenchmarkCommand :
  ↪ ICommand
{
  private WriteBenchmarkCommand
    ↪ _writeBenchmark = new
    ↪ WriteBenchmarkCommand();
  private ReadBenchmarkCommand
    ↪ _readBenchmark = new
    ↪ ReadBenchmarkCommand();

  // Other Fields
  // Other Methods

  public async Task Stop()
  {
    await _writeBenchmark.Stop();
    await _readBenchmark.Stop();
  }
}
src/Commands/TestBenchmarkCommand.cs:
[TestMethod]
public void TestStop()
{
  var command = new
    ↪ BenchmarkCommand();
  command.Stop().Wait();
  Assert.IsTrue(command.IsStopped());
}
src/Commands/TestBenchmarkCommand.cs:
[TestMethod]
public void TestBenchmarkCommand()
// More generated text...
```

Filepath · Adaptive Focal Context · Focal Method · Hint · Generated test · Extra text · Prompt · Completion

Fig. 1: Our prompt design for test generation. The context given to the model is truncated to fit within the model's context length.

We employ *adaptive focal context* [47] to ensure the inputs fit within the model's context length, trying first the entire focal file, then abbreviating context methods, fields, and comments. To generate diverse test suites, we set $temperature$ = 0.7, top_p = 1.0, and $frequency\_penalty$ = 0.5 and generated multiple completions for each prompt. For complete details, see the online appendix [2],

C. Evaluating Test Case quality

We assessed the properties of the generated unit tests with respect to commonly-accepted best practices (e.g. syntactic correctness) and test-smells (e.g. missing assert statement, missing call to the focal method, or conditional logic); see Section III-A for the full list of metrics. We implemented a tool which we call *Quality Analyzer* to evaluate these quality metrics. Inspired by tsDetect for Java [31], our tool uses *tree-sitter* [1] to traverse the Abstract Syntax Tree (AST) and analyze the code's quality properties. We used Quality Analyzer to automatically provide the feedback needed to construct a dataset for SFT and perform RL training. This tool is available in our data package [2].

D. Supervised Fine-tuning

Figure 2 (❶) illustrates the supervised fine-tuning (SFT) process. We used the Quality Analyzer to create a *golden dataset* for supervised fine-tuning. We filtered the unit test to exhibit syntactic correctness, contain at least one assertion, and invoke the focal method, while avoiding duplicate assertions and conditionals. We then fine-tune the language model to generate these high-quality tests by minimizing the cross-entropy loss.

By filtering the dataset to only include test cases with desirable qualities, we are effectively training a *refined* version of the model which generates higher-quality test cases. Although this approach consistently produced higher-quality test cases, our experimental results (discussed in Section IV-A) highlight some limitations of this approach, including the fact that it cannot learn from negative examples. In contrast, RL enables the model to actively penalize undesirable behaviors and test smells, offering a more robust training process.

E. Reinforcement Learning from Static Quality Metrics

We frame reinforcement learning fine-tuning as an episodic RL scenario [44]. In each episode, the agent generates a unit test case by sequencing tokens from the vocabulary of tokens. Each action generates a single token, each observation consists of the prompt + the sequence of tokens generated in previous steps, and the episode ends when the model generates a *stop* token or reaches the maximum number of tokens. The agent relies on a policy π, instantiated by a pretrained language model, to decide its next action. This policy seeks to guide the agent in making decisions that maximize the episodic reward. At the end of the episode, the agent receives a reward $\mathcal{R}$ which corresponds to the successful completion of a unit test case.

1) Reward Model Training: Figure 2 (❷) zooms in on the RLSQM reward model (RM) finetuning stage. As in RLHF, the reward model is pivotal to our approach. However, rather than relying on human feedback to train the RM, we employ a static Quality Analyzer to automatically determine a scalar reward score for each prompt and completion (see § II-F for details of the reward score). These derived rewards are then harnessed to train a model to estimate expected rewards from the agent's decisions.

The RM is trained by optimizing the Mean Squared Error (MSE) loss, given by:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(R(x_i, y_i) - \hat{R}(x_i, y_i))^2 \qquad (1)$$

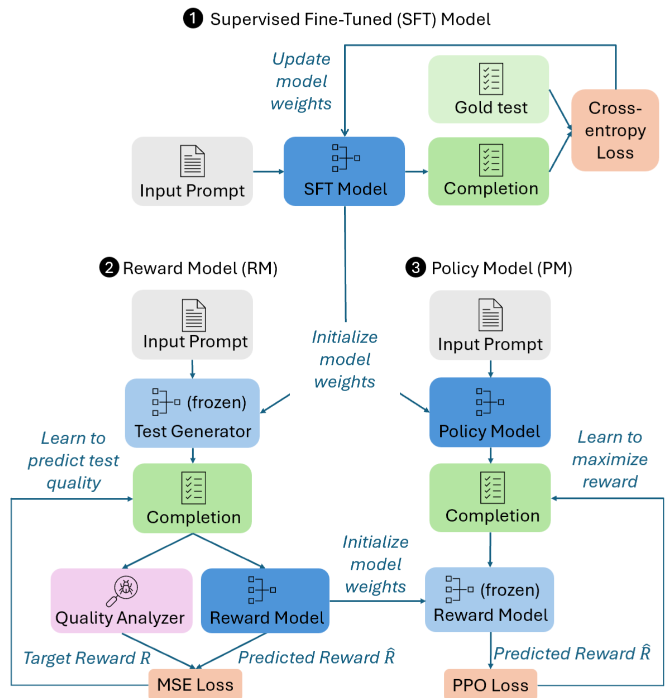

Fig. 2: Overview of our approach: Reinforcement Learning from Static Quality Metrics (RLSQM). (1) We automatically generate tests and filter them to a *golden set* of high-quality of using Quality Analyzer, then train the model to generate high-quality tests with supervised fine-tuning (SFT). (2) We generate tests with the SFT model and use Quality Analyzer to train the reward model (RM) to predict accurate reward scores. (3) We initialize the policy model (PM) with the SFT model weights and fine-tune it with PPO to maximize the rewards predicted by the RM.

In this equation, $R(x_i, y_i)$ is the reward for one generated test y_i with prompt x_i as per the Quality Analyzer, and $\hat{R}(x_i, y_i)$ is its predicted counterpart.

The RM learns to predict a scalar reward score that closely matches the Quality Analyzer. While the Quality Analyzer could directly provide scores suitable for RL fine-tuning, we note that the RM is more general and can easily extend to less-defined rewards, including human feedback; it can learn to combine multiple quality metrics; and it can estimate a reward score even when static analysis tools fail (incorrect syntax or partial code snippets).

2) PPO Fine-tuning: Figure 2 (❸) displays the RLSQM's policy model (PM) finetuning stage. RLSQM utilizes the Proximal Policy Optimization (PPO) [37] algorithm to train the PM. We initialized the policy π using the SFT model and initialized the value function using the weights of the reward model (following [26]). The PM learns to optimize the reward, minus a penalty when it generates tokens that are dramatically different from the base model (also following [26]):

$$\mathcal{R}(x, y) = \left[\hat{R}(x, y) - \beta D_{\text{KL}}(\pi_{\theta_0}, \pi_\theta)\right] \qquad (2)$$

Here, π_{θ_0} represents the initial weights of the policy before PPO fine-tuning, π_θ represents the policy's current weights, D_{KL} denotes the function computing the KL divergence, and and β is a hyperparameter scaling the KL divergence penalty.

The policy model is trained to maximize $\mathcal{R}(x,y)$ by minimizing the clipped surrogate objective:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)A^{\text{old}}, \text{clip}_{1\pm\epsilon}\left(r_t(\theta)\right)A^{\text{old}}\right)\right] \quad (3)$$

Where the ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ contrasts the new policy's probability to the old's for action a_t in state s_t, A^{old} is the advantage estimate on state-action pair (s_t, a_t) from the old policy, which is computed based on $\mathcal{R}$, and ϵ is a PPO clipping parameter that confines the policy's update magnitude.

Our approach is not tied to the PPO algorithm; implementations may employ a variety of other RL algorithms, such as A2C [22] or NLPO [32].

F. Training strategies for RLSQM

Our approach of training the reward model is general and therefore, many possible strategies exist for selecting the target reward function (R). We propose three training strategies which assign negative reward to discourage syntactically-incorrect completions and grant positive rewards based on quality properties.

Individual Reward: In the most basic setting, the target reward function, denoted $R_P(x_i, y_i)$ for some prompt and completion x_i, y_i and property P, is assigned based on whether the test is syntactically correct and exhibits P, according to Table II (left). The resulting reward model provides higher rewards to test cases that adhere to the chosen quality metric during the PPO fine-tuning phase.

TABLE II: Target reward (R): Individual Reward for a property P (left) and Combined Reward for 2 properties P and Q (right).

Individual Reward

Correct Syntax	P	Reward
False	$*$	-1
True	False	0
True	True	1

Combined Reward

Correct Syntax	P	Q	Reward
False	$*$	$*$	-1
True	False	False	0
True	True	False	1
True	False	True	1
True	True	True	2

Sequential & Combined Reward: We also integrate multiple quality metrics, aiming to train a reward model which models the overall quality of generated tests among several metrics. We expect that a model which optimizes this learned reward will generate high-quality test cases according to multiple metrics.

We developed two strategies for training the reward model. In the first strategy, called **Sequential Reward**, we take the policy model trained with Individual Reward (now called the 1st stage model) and re-train it to optimize a different quality metric; we call this the 2nd stage model. Our goal is to preserve the relative simplicity of individual reward training in each stage while optimizing the 2nd-stage model for additional properties.

The second strategy, called **Combined Reward**, combines multiple target reward functions. For each test, we analyze the properties and assign the rewards for two properties individually, then add the reward values from the two functions. This yields the composite reward function shown in Equation (4), defined over k quality properties. An example for two properties is shown in Table II (right).

$$R_{\text{Comb}}(x_i, y_i) = \begin{cases} \sum_{j=1}^{k} R_{\text{P}_j}(x_i, y_i) & \text{if } y_i \text{ has correct syntax;} \\ -1 & \text{otherwise.} \end{cases} \quad (4)$$

III. Study Design

We designed an empirical study with the primary objective of enhancing the quality of test cases generated by Language Model-based approaches through Reinforcement Learning from Static Quality Metrics (RLSQM). Our study aims to address the following research questions:

- RQ_1: What is the quality of the test cases generated by Codex?
- RQ_2: Can RLSQM improve Codex to generate high-quality tests?

A. RQ_1: Assessing Test Case Quality

To address RQ_1, we generated test cases for a diverse set of focal methods. We generated 100 tests per focal method, then automatically analyzed the test cases, focusing on crucial properties introduced by Peruma et al. [30], shown in Figure 3.

Based on the results, we selected the quality metrics that required the most improvement in the base LLM. We focus our evaluation on statically-detectable qualitative metrics; dynamic metrics (such as the rate of compilability, passing tests, or coverage) have been studied in other research [47, 3, 19] and are left to future work.

B. RQ_2: Evaluating RLSQM's Effectiveness

To investigate RQ_2, we compared RLSQM's best-performing training strategy with state-of-the-art code generation models: GPT-4, Base Codex model, and Codex with Supervised Fine-Tuning (SFT) on gold datasets, as proposed in Section II-D. We evaluated all models based on the syntactic correctness and the frequency of quality properties in the tests generated for a held-out set of focal methods. We performed several ablations to find the best-performing setting for RLSQM, including training with Individual, Combined, and Sequential rewards.

IV. Experiments and Results

A. Experimental settings

Training Procedure: We held out 5% of focal methods for testing and used the remainder for training, sampling the training and test data from different repositories to prevent data leakage. For all training settings (SFT, RM and PM), we held out 10% validation data by repository. We found that validation was critical to avoid mode collapse [8] and catastrophic forgetting. During SFT and RM training, we

Fig. 3: Test quality properties. ✓ marks desirable properties and ✗ marks test smells.

✓ Necessary Test Properties

- **Correct Syntax:** All test cases should be syntactically valid according to the C# language specification.

✓ Best Practices

- **Contains Assertion:** Each test case should contain at least one assertion. Tests without assertions can be useful; however, when an assertion is present, a developer can observe its conditions to reason about the purpose of the test.
- **Invokes Focal Method:** Each test case should call the focal method which is specified in the prompt, in order to produce tests relevant to a developer's request.

✓ Documentation

- **Includes Comment:** The presence of at least one comment, enhancing test case documentation.
- **Has Descriptive Name:** The presence of additional text in the name of the test, which can act as documentation. For example, a name stub `TestAdd` may be enhanced by expanding it to `TestAdd_EmptyString_ReturnsZero`, documenting the input and purpose of the test [35].

✗ Test Smells [30]

- **Contains Duplicate Assertion:** Test cases should not contain consecutive identical assertion statements, which may be redundant or inefficient.
- **Contains Conditional Logic or Exception Handling:** Test cases containing conditional statements (such as `if`, `switch`, `while`) and exception handling (such as `try`/`catch` blocks) are more complex to maintain and reason about.

performed early stopping based on the lowest loss on validation data; during PM training, we performed early stopping based on the RM's predicted quality score. We ensured that all reward models converged to near-0 MSE loss on validation data and all policy models increased the modeled reward $\mathcal{R}$. Because the RM training data was derived from the Base model's tests, the representation of different test smells was imbalanced; to remedy this bias, we resampled the tests, training the RM on a balanced label distribution. For complete details of our training procedure, see the online appendix [2].

Base model: We generate tests using the OpenAI Codex Cushman model, version `code-cushman-001`. This version of Codex is a GPT-style model, pretrained on 54 million public repositories using the causal language modeling objective.

Supervised fine-tuning: RLSQM was unstable during training when directly applied to the Base model, generating repetitive assertions or comments, or empty tests. To mitigate these behaviors, we initialized RL fine-tuning with the supervised fine-tuned model, following prior literature [43, 26].

Codex generated Documentation properties relatively infrequently (§ IV-B), resulting in a 50x smaller SFT training set on intersection with the other properties. The resulting model, denoted SFT_{Doc}, improved on Comments and Descriptive Names but regressed on all five of the other properties. This

TABLE III: Codex-generated test quality on the entire corpus of C# focal methods. Frequency denotes the proportion of generated tests containing each smell. Test smell priority is based on developer studies [38, 42].

Property	Frequency	Optimization Target
✓ Necessary Properties		
Correct Syntax	83.49%	-
✓ Best Practices		
Contains Assertion	69.18%	Yes
Calls Focal Method	63.44%	Yes
✓ Documentation		
Descriptive Name	12.70%	Yes
Contains Comment	20.71%	Yes
✗ Test Smells		
Duplicate Assertion	2.26%	Yes
Conditional/Exception	2.31%	Yes
Assertion Roulette	18.11%	No (low-priority)
Magic Number	17.78%	No (low-priority)
Sensitive Equality	2.65%	No (low-priority)
Redundant Print	1.40%	No (infrequent)
Sleepy Test	0.60%	No (infrequent)
Empty Test	0.34%	No (infrequent)
Mystery Guest	0.24%	No (infrequent)
Resource Optimism	0.01%	No (infrequent)

highlights a limitation of filtering-based SFT – properties with low frequency or few co-occurrences restricted the size of the golden training dataset. We trained another SFT model filtered only on Best Practices and Test Smells, which we simply call *SFT*; it regressed on Documentation properties but improved all others, so we used it to initialize all instances of RLSQM.

GPT-4 baseline: We compare with GPT-4 [25] (version `gpt-4-0613`) prompted to generate tests. Although RLSQM is applicable to any LM, GPT-4 does not support RL fine-tuning. We generated one candidate test for each focal method due to the increased cost and used the same settings for all generation hyperparameters. Compared to our simulated prompt, GPT-4 produced more syntactically-correct tests when prompted with natural-language instructions (`Write a C# unit test using MSTest for the method ⟨FocalMethod⟩. Output executable code only.`), followed by the focal code and the prompt hint. We used this prompt to elicit the best performance from GPT-4 and applied adaptive focal context to the focal code to ensure a fair comparison.

We trained the models on Azure ML, utilizing virtual machines equipped with 8 NVIDIA A100 GPUs, 96 vCPUs, and 1,924GB RAM. Each GPU had 40GB of HBM2 device memory. Supervised training took about 16 hours. Reward model training took about 2.5 hours and policy model training took about 4 hours, per run.

B. RQ_1: Assessing Test Case Quality

Codex-generated tests often lacked critical quality properties or contained test smells, both of which contribute to the maintainability and understandability of the test suite. Table III

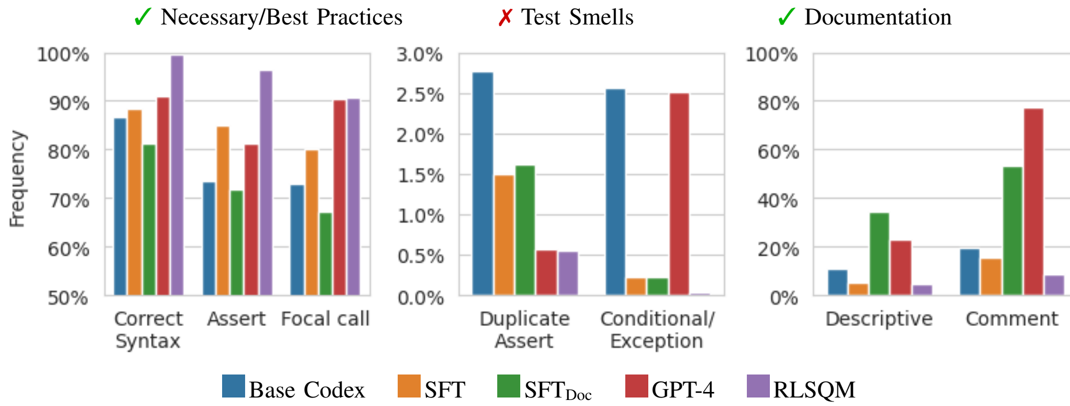

Fig. 4: Comparison between RLSQM's best configuration and baselines. RLSQM outperformed all baselines on all Best Practices and Test Smells, but underperformed on Documentation. [1]

lists the frequencies of the properties we computed on the tests generated by the base model, organized in the sub-categories: Necessary, Best Practices, Documentation, and Test Smells. 83% of the base model's completions were syntactically correct, with the remaining 17% containing syntax errors. Up to 37% of generated tests were lacking assertions and 31% were lacking calls to the focal method, showing room for improvement. Regarding documentation, the base model generated a small percentage of tests containing comments (20.7%) and descriptive names (12.7%), marking these as desirable properties for optimization.

Out of the test smells outlined in Peruma et al. [30], we chose to optimize only Duplicate Assertion and Conditional/Exception because we believe that these are most impactful to the developer. Given the large number of experiments required to study the smells in-depth, we had to be selective in the number of properties to optimize. Six of the test smells not shown were not applicable in our single-test-generation setting. We regarded the five test smells which occurred in less than 2% of tests as subpar optimization targets. Developers considered Conditional Test Logic to have Very High Impact on maintainability in 19.4% of responses, compared to only 2.1% for Assertion Roulette and 5.0% for Magic Number Test ([42] Table 7). In a prior study, developers considered Sensitive Equality as a false positive or no-action item [38]; additionally, this smell can easily be refactored away automatically [46]. We consider Duplicate Assertions as likely to confuse the developer by generating redundant code, being affected by side-effects, and obscuring the test's passing conditions.

In Table IV, we also compare our test smells with the results reported on SF110 [40], a set of open-source Java projects similar to our C# dataset. While both papers include several test smells not shown, we report the test smells in common on the Codex-2k model. Siddiq et al. reported similar frequencies for Unknown Test (UT, the inverse of Contains Assertion), Duplicate Assertion (DA), and Conditional Logic Test (CLT).

[1]Base model results vary slightly from Table III because RQ_1 evaluates Base Codex on the entire dataset while RQ_2 & RQ_3 evaluate models on a held-out subset (§IV-A).

TABLE IV: Codex-generated test quality on C# focal methods versus Java methods reported by Siddiq et al. [40].

Dataset	UT	DA	CLT**	EH**
SF110 (Java)	21.20%	1.40%	0.50%	20.70%
Ours (C#)	26.64%	2.77%	2.56%	2.56%

**We merged CLT and EH into one test smell.

Exception Handling (EH) occured more frequently in the Java programs, which we believe is due to a difference between the languages; Java programs will not compile if a checked exception is not caught, so the Codex model may tend to catch exceptions inside the test in order to more easily produce a compilable program.

C. RQ_2: Effectiveness of RLSQM

We ran RLSQM to fine-tune models for each individual quality property, then for combinations of properties. Figure 4 shows the performance of the best-performing RLSQM approach (✓ *Focal* + ✗ *Conditional/Exception*) compared with baselines. RLSQM substantially outperformed all baselines on all Necessary/Best Practices and Test Smells, improving upon the Base model by up to 23.2%, SFT by up to 11.6%, SFT_{Doc} by up to 24.7%, and GPT4 by up to 15.3%. Notably, RLSQM produced next to no tests with Conditionals/Exceptions (0.03%), nearly all its tests were syntactically-correct (99.6%, compared to 90% in the best-case for GPT-4). These results clearly demonstrate that RLSQM is an effective method which substantially improved the quality of tests generated by Codex. We believe this is enabled by incorporating negative rewards from syntactically-incorrect code, which give policy model direct feedback against generating any syntax errors.

GPT-4, the strongest baseline, produced tests with more Descriptive Names (23%) and more Comments (78%) than RLSQM. We hypothesize that the initial SFT phase of RLSQM, which introduced regressions on Documentation properties, constrained the policy model's ability to enhance these specific properties. Further research is needed to remove this limitation inherited from the SFT stage. Since GPT-4 is trained to understand and generate human-like code [25], it's not surprising that it produced documentation and avoided consecutive duplicated assertions. Considering the practical dimension of deployment cost, we estimate that fine-tuned Codex (12B parameters) is several orders of magnitude smaller and costs less than half as much as GPT-4 [2], and thus RLSQM can be deployed to generate less-smelly tests at lower cost. For details of our ablation experiments on optimizing multiple quality properties, see the online appendix [2].

V. Related Work

Previous research on unit test generation has employed evolutionary algorithms, leading to tools such as EvoSuite [15] and Randoop [27], and several notable machine learning models [47, 3, 19, 13, 40, 39, 49, 51, 12, 28, 14, 34, 18]. The aim of these efforts is generally to resolve compiler errors, produce passing tests from the current code, generate failing tests to expose bugs, or boost code coverage, while we focus on code quality such as test smells, orthogonal qualities which are important for usability and maintainability. Most closely related, Siddiq et al. [40] demonstrated that ChatGPT and Codex are prone to produce test smells on Python and Java code (see § IV-B for a comparison with our findings); however, they do not suggest how to improve the models. In this study, we introduce RLSQM as a method to *enhance* language models based on static quality metrics. In addition, compared to the existing works, we (1) propose SFT fine-tuning based on filtering gold tests followed by an RL stage, (2) introduce LLM-based reward models, and (3) provide practical guidance about applying RL to LLMs for test generation.

VI. Threats to Validity

We selected from a large set of 100 projects, including projects from the domains of cloud computing, telecommunications, and game engines, showing that our dataset is indeed diverse. Furthermore, we extracted the public methods which were targeted by existing tests, with the effect of studying the methods the developer intended to test at some point. This has the added effect of requiring the projects included in the dataset to contain unit tests, which further culls possible toy projects. Data leakage could occur due to our method of holding out the evaluation dataset. We mitigated this concern by ensuring that the training and test dataset come from different projects, and thus are less likely to share code.

Our experimental results focus on the C# language due to the availability of our dataset, but we did not evaluate RLSQM on other popular languages like as Java or Python. However, our approach is not restricted to C#; it can be extended to other languages by re-implementing the simple AST traversals in the Quality Analyzer [2]. C#, like Java, is object-oriented, and both languages have a strong emphasis on unit testing. As observed in Section IV-B, models produced similar test smell distributions in both languages.

Our choice of model and sampling method is inherently non-deterministic, and thus produces different results when run multiple times. We accounted for this variance by generating 100 tests for each method and averaging the results. Our results also depend on the choice of hyperparameters. We performed manual hyperparameter tuning on the Individual Rewards model to get the best-performing configuration within several settings of the key hyperparameters.

VII. Conclusion

Reinforcement learning has seen great success in aligning base language models with user intents. In this paper, we propose Reinforcement Learning from Static Quality Metrics (RLSQM), a fully-automated approach wherein we train a reward model on static analysis-based quality metrics, then optimize a large language model to maximize those rewards. Our fine-tuning approach involves a basic application of Supervised Fine-tuning, plus an exploration of RL training strategies involving optimizing one or multiple rewards. We also report how to effectively train models with RLSQM based on our experience and experimental findings. Our evaluation shows that RLSQM substantially improved the quality of tests generated by Codex, and produced higher-quality tests than GPT-4 at a lower cost.

[2]Up-to-date pricing information is no longer available for code-cushman-001 (12B parameters); however, at less than one-tenth the size, we conservatively estimate that RLSQM will cost even less than fine-tuned davinci-002 (175B) at $0.012/KT (thousand-tokens), which costs less than half of GPT-4 at $0.03/KT (https://openai.com/pricing).

References

[1] tree-sitter: An incremental parsing system for programming tools. http://tree-sitter.github.io/tree-sitter.

[2] Data package, 2024. URL https://doi.org/10.6084/m9.figshare.25983166. Figshare.

[3] Saranya Alagarsamy, Chakkrit Tantithamthavorn, et al. A3Test: Assertion-augmented automated test case generation. arXiv, 2023.

[4] Yuntao Bai, Andy Jones, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. arXiv, 2022.

[5] Yuntao Bai, Saurav Kadavath, et al. Constitutional ai: Harmlessness from ai feedback. arXiv, 2022.

[6] Gabriele Bavota, Abdallah Qusef, et al. Are test smells really harmful? an empirical study. *ESE*, 2015.

[7] Samy Bengio, Oriol Vinyals, et al. Scheduled sampling for sequence prediction with recurrent neural networks. In *NeurIPS*, 2015.

[8] Stephen Casper, Xander Davies, et al. Open problems and fundamental limitations of reinforcement learning from human feedback. arXiv, 2023.

[9] Aaron Chan, Anant Kharkar, et al. Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning? arXiv, 2023.

[10] Mark Chen, Jerry Tworek, et al. Evaluating large language models trained on code. arXiv, 2021.

[11] Paul Christiano, Jan Leike, et al. Deep reinforcement learning from human preferences. arXiv, 2023.

[12] Arghavan Moradi Dakhel, Amin Nikanjam, et al. Effective test generation using pre-trained large language models and mutation testing. arXiv, 2023.
[13] Elizabeth Dinella, Gabriel Ryan, et al. Toga: A neural method for test oracle generation. In *ICSE*, 2022. doi: 10.1145/3510003.3510141.
[14] Mehdi Esnaashari and Amir Hossein Damia. Automation of software test data generation using genetic algorithm and reinforcement learning. 2021. doi: 10.1016/j.eswa.2021.115446.
[15] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE*, 2011.
[16] GitHub. GitHub Copilot, 2021.
[17] Daya Guo, Alexey Svyatkovskiy, et al. Learning to complete code with sketches. In *ICLR*, 2021.
[18] Jinkyu Koo, Charitha Saumya, et al. PySE: Automatic Worst-Case Test Generation by Reinforcement Learning. In *ICST*, 2019. doi: 10.1109/ICST.2019.00023.
[19] Caroline Lemieux, Jeevana Priya Inala, et al. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *ICSME*, 2023. doi: 10.1109/ICSE48619.2023.00085.
[20] Haokun Liu, Derek Tam, et al. Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning. 2022.
[21] Microsoft. Unit testing C# with MSTest and .NET. https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest, 2023.
[22] Volodymyr Mnih, Adrià Puigdomènech Badia, et al. Asynchronous methods for deep reinforcement learning. arXiv, 2016.
[23] Liang Niu, Shujaat Mirza, et al. CodexLeaks: Privacy leaks from code generation language models in GitHub copilot. In *USENIX*, 2023.
[24] OpenAI. Introducing chatgpt. https://openai.com/index/chatgpt/, 2023.
[25] OpenAI. Gpt-4 technical report. arXiv, 2023.
[26] Long Ouyang, Jeffrey Wu, et al. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.
[27] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA*, 2007.
[28] Zhixin Pan and Prabhat Mishra. Automated Test Generation for Hardware Trojan Detection using Reinforcement Learning. In *ASPDAC*, 2021. doi: 10.1145/3394885.3431595.
[29] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub copilot's code contributions. In *IEEE S&P*, 2022.
[30] Anthony Peruma, Khalid Almalki, et al. On the distribution of test smells in open source Android applications: an exploratory study. In *CASCON*, 2019.
[31] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsDetect: an open source test smells detection tool. In *ESEC/FSE*, 2020.
[32] Rajkumar Ramamurthy, Prithviraj Ammanabrolu, et al. Is reinforcement learning (not) for natural language processing: Benchmarks, baselines, and building blocks for natural language policy optimization. arXiv, 2023.
[33] Marc'Aurelio Ranzato, Sumit Chopra, et al. Sequence level training with recurrent neural networks. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.
[34] Sameer Reddy, Caroline Lemieux, et al. Quickly generating diverse valid test inputs with reinforcement learning. In *ICSE*, 2020. doi: 10.1145/3377811.3380399.
[35] John Reese. Best practices for writing unit tests - .NET. https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices#naming-your-tests, 2022.
[36] Baptiste Rozière, Jonas Gehring, et al. Code llama: Open foundation models for code. arXiv, 2023.
[37] John Schulman, Filip Wolski, et al. Proximal policy optimization algorithms. arXiv, 2017.
[38] Martin Schvarcbacher, D. Spadini, et al. Investigating developer perception on test smells using better code hub. 2019.
[39] Max Schäfer, Sarah Nadi, et al. An empirical evaluation of using large language models for automated unit test generation. arXiv, 2023.
[40] Mohammed Latif Siddiq, Joanna C. S. Santos, et al. Exploring the effectiveness of large language models in generating unit tests. arXiv, 2023.
[41] Davide Spadini, Fabio Palomba, et al. On the relation of test smells to software code quality. In *ICSME*, 2018.
[42] Davide Spadini, Martin Schvarcbacher, et al. Investigating Severity Thresholds for Test Smells. In *MSR*, 2020. doi: 10.1145/3379597.3387453.
[43] Nisan Stiennon, Long Ouyang, et al. Learning to summarize from human feedback. In *NeurIPS*, 2020.
[44] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[45] Hugo Touvron, Louis Martin, et al. Llama 2: Open foundation and fine-tuned chat models. arXiv, 2023.
[46] Michele Tufano, Fabio Palomba, et al. An empirical investigation into the nature of test smells. In *ASE*, 2016.
[47] Michele Tufano, Dawn Drain, et al. Unit test case generation with transformers and focal context. arXiv, 2021.
[48] Arie Van Deursen, Leon Moonen, et al. Refactoring test code. In *XP2001*, 2001.
[49] Zhuokui Xie, Yinghao Chen, et al. Chatunitest: a chatgpt-based automated unit test generation tool. arXiv, 2023.
[50] Shunyu Yao, Dian Yu, et al. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv, 2023.
[51] Zhiqiang Yuan, Yiling Lou, et al. No more manual tests? evaluating and improving chatgpt for unit test generation. arXiv, 2023.
[52] Daniel M. Ziegler, Nisan Stiennon, et al. Fine-tuning language models from human preferences. arXiv, 2020.