Unit Test Generation using Large Language Models for Unity Game Development

Ciprian Paduraru
University of Bucharest
Romania
ciprian.paduraru@unibuc.ro

Alin Stefanescu
University of Bucharest
Romania
Institute for Logic and Data Science
Romania
alin.stefanescu@unibuc.ro

Augustin Jianu
certSIGN
Romania
augustin.jianu@certsign.ro

ABSTRACT

Challenges related to game quality, whether occurring during initial release or after updates, can result in player dissatisfaction, media scrutiny, and potential financial setbacks. These issues may stem from factors like software bugs, performance bottlenecks, or security vulnerabilities. Despite these challenges, game developers often rely on manual playtesting, highlighting the need for more robust and automated processes in game development. This research explores the application of Large Language Models (LLMs) for automating unit test creation in game development, with a specific focus on strongly typed programming languages like C++ and C#, widely used in the industry. The study centers around fine-tuning Code Llama, an advanced code generation model, to address common scenarios encountered in game development, including game engines and specific APIs or backends. Although the prototyping and evaluations primarily occurred within the Unity game engine, the proposed methods can be adapted to other internal or publicly available solutions. The evaluation outcomes demonstrate the effectiveness of these methods in enhancing existing unit test suites or automatically generating new tests based on natural language descriptions of class contexts and targeted methods.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Development frameworks and environments**.

KEYWORDS

large language models, unit testing, game development

ACM Reference Format:
Ciprian Paduraru, Alin Stefanescu, and Augustin Jianu. 2024. Unit Test Generation using Large Language Models for Unity Game Development. In *Proceedings of the 1st ACM International Workshop on Foundations of Applied Software Engineering for Games (FaSE4Games '24), July 16, 2024, Porto de Galinhas, Brazil*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663532.3664466

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FaSE4Games '24, July 16, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0674-5/24/07
https://doi.org/10.1145/3663532.3664466

1 INTRODUCTION

Ensuring the quality of video games and integrating effective unit testing into the game development process presents several challenges. Research studies, including the one cited by [21], consistently reveal that games regardless of budget or team size frequently launch with bugs. These issues can result in severe criticism and potential financial setbacks if the game fails to meet user expectations. A significant portion of these quality problems can be attributed to insufficient testing practices. Surprisingly, game developers often heavily rely on manual playtesting and the experiential insights of human testers. While automation is recognized as a crucial step toward enhancing game quality, existing testing techniques may not universally apply to all game types. In some high-profile projects, developers even operate without a single unit test. Furthermore, most developers specialize in specific areas of a project, emphasizing the need for more robust and automated testing procedures across the game development lifecycle. However, the unique characteristics of games—such as interactivity and immersion—pose challenges when implementing traditional software testing methods. Unit testing plays a pivotal role in software development by scrutinizing individual code segments in isolation. Detecting and rectifying bugs early in the development cycle is essential, as it provides insights into how different code components within a software system interconnect and function as a cohesive whole. Despite its importance, developers often face obstacles when creating unit tests, leading to inadequate test coverage. Surprisingly, an earlier study examining 82,447 GitHub projects found that only 17% included test files [10].

As Large Language Models (LLMs) gain prominence in code generation, researchers have explored their accuracy in producing code [8], code quality [24], security implications [20], and their potential for API learning and code complexity prediction. However, how these apply to game development and engines remains uncertain. Challenges arise due to limited training data availability, the strict type-checking inherent in strongly typed languages, and the difference in the APIs and terminologies used. This study aims to explore the capabilities of LLMs to automate unit test creation for game development. We start with Code Llama [22] as our foundational model and further fine-tune it using a dataset extracted from game development examples.
Our contribution can be summarized as follows:

- As far as our knowledge extends, this study represents the first exploration of using Large Language Models (LLMs) to create unit tests specifically for game development, regardless of the

game engine, and considering both statically typed languages, C#, C++.
- We introduce an automatic generating corpora method, LLM based, for generating synthetic datasets of unit-tests for training models or evaluation purposes. The goal is to remove the bottleneck of not having enough data, or the costs needed to get one from human professionls.
- A human-in-the-loop method for correcting and guiding unit-tests generation towards improved code coverage.
- The goals are defined after discussion with the local game development industry. The experiments are made with two publicly available game engines, Unity[1], and Unreal[2] which are used by companies, or resemble most of commonalities with private one in companies.
- Our work involves adapting LLM-related processes—such as fine-tuning, prompting, and retrieval augmented generation (RAG) [17]—to align with typical game development scenarios. The prototype solution is offered as open source code at https://github.com/unibuc-cs/GameTestingWithLLMs.

The subsequent sections of this article are structured as follows. We begin by discussing related work aligned with our research objectives. Section 3 presents the process of extracting a dataset specific to game development source code. Section 4 outlines the methods and procedures employed for sampling, filtering, and fine-tuning the foundational model. Our evaluation results and potential enhancements are presented in the dedicated evaluation section. Lastly, we conclude with insights in the final section.

2 RELATED WORK

Difficulties in Unit Testing. The process of manually developing and maintaining top-tier unit tests can be a daunting and lengthy task [26]. To alleviate the manual effort required in crafting unit tests, various automated unit test generation methods have been proposed by researchers. Conventional methods for creating unit tests typically employ search-based [4], constraint-based [1], or random-based strategies [5], [6]. The primary objective of these techniques is to produce a suite of unit tests that optimize the coverage of the software being tested. While these auto-generated tests attain a satisfactory coverage level, they often fall short in terms of readability and expressiveness when compared to tests written manually. As a result, developers are usually hesitant to directly integrate them into their workflow [26].

LLMs in Code Generation. Numerous language learning models like CodeBert [9], CodeGen [19], CodeT5 [28], Llama2-Code [22] have been made available for code generation after undergoing fine-tuning on extensive code datasets. Following this, GitHub Copilot introduced an advanced autocomplete feature to assist with basic algorithmic problems, utilizing an enhanced version of Codex [7]. Our research does not concentrate on code generation, but rather on how LLMs can be fine-tuned and adapted for unit test generation using a zero-shot prompt, with the potential to incorporate existing codebases from game development APIs and infrastructures.

LLMs in Unit Test Generation. The study in [29] is an empirical assessment of the capability of Large Language Models (LLMs) to generate unit tests without additional training or manual intervention for Java code, comparing the outcomes between three public models: Codex[7], GPT-3.5-Turbo, and StarCoder [13]. The authors initially investigate the impact of the context provided as a prompt on the unit test generation process. The models are evaluated on several metrics: compilation rates, test accuracy, test coverage, and test smell. A similar objective is outlined in [25], where the authors evaluate ChatGPT, this time to assess the generated unit tests. Our approach differs from their work in that we implement a fine-tuning method to tailor the base LLM model to comprehend the code of the basic functions of the game engine. We then compare the unit tests generated by the foundational LLM models and the fine-tuned models using the same metrics.

The TestGen-LLM tool, developed by Meta [2], leverages the Code Llama model [22] to enhance pre-existing tests crafted by software engineers. This tool refines test cases automatically and validates the enhanced versions using a sequence of filters, ensuring a notable enhancement over the original test suite. It also tackles issues related to LLM hallucinations. TestGen-LLM has been effectively implemented on Meta's platforms, including Instagram and Facebook, where it boosted the quality of existing unit test sets by 11.5%, as per the report. Moreover, a substantial fraction of these generated tests, 73%, received approval for production from Meta's software engineers.

3 GATHERING A CORPUS

The motivation to gather a training and evaluation corpus along this projects stems from two main observation made after an initial research of existing datasets of unit tests:

(1) Limited resources for languages such as C++ and C#.
(2) The lack of datasets for game development. This is an important aspect because sometimes the terminology used and APIs can differ from the general programming cases.

What is specific to game development is that the source code frequently depends on its foundation engine components (e.g., the ones in Unity on Unreal Engine), or project internal libraries and APIs. During testing, these components are mocked, allowing the tests to run independently, as necessitated by the standard methodology of unit testing. Both training and evaluation corpora should prepare (fine-tune) the core LLM such that it comprehends these backend functions, their components, classes, methods, and interfaces.

Our method tries first to collect unit tests from publicly available projects. Using this dataset, which we denote by $\mathcal{D}_h$, a preliminary model, named $GameUnitLLM_{v0}$ is trained to produce unit tests. Based on this initial version, we build a synthetic method to build automatically, without human annotations a dataset of valid unit-tests. We name this dataset $\mathcal{D}_s$, thus the total corpora is represented by the union of the two, Eq: 1.

$$\mathcal{D} = \mathcal{D}_h \cup \mathcal{D}_s \tag{1}$$

$\mathcal{D}$ this is further divided into two subsets, a training and a testing one, i.e., $\mathcal{D} = \mathcal{D}_{test} \cup \mathcal{D}_{train}$. For evaluation purposes and comparisons between the two programming languages used typically in game development, we split these sets for unit-tests in *C#*,

[1] www.unity.com
[2] www.unrealengine.com

respectively $C++$: D^{cs}, and D^{cpp}. Further, fine-tuning on $\mathcal{D}_train$, produces the final targeted LLM model $GameUnitLLM$. A separate fine-tuning is made using the two splits, producing models $GameUnitLLM^{cs}$, respectively $GameUnitLLM^{cpp}$. The two steps are explained in the continuation of this section, and can be visualized in Figure 1.

3.1 Dataset of Unit-Tests Written by Humans

First, nine project implementations that were open-sourced are gathered as follows:

- Five projects from the Unity Learn platform[3] and their corresponding tasks solutions examples from GitHub repositories.
- The four (at the moment of writing) educational Unreal Engine's game samples on their marketplace platform [4].

From these projects, we pinpoint their first layer of dependencies and extract their source code, i.e., a collection of both C++ and C# script files, where the source code was publicly available. Extracted package included functions and APIs specifically for game development such as input processing, animation, pathfinding, physics, UI-related packages, etc.

From the collected set of projects, the source code corresponding to the unit-tests is exctracted automatically by traversing the scripts and picking out the methods that carry a particular attribute according to either Unity/Unreal backend implementation (e.g., *[TEST]* in Unity, or *[...AUTOMATION_TEST]* in Unreal). These methods are then grouped by class and we compile all the available code for class definition (Cls) along with the corresponding discovered unit tests, that is, pairs of the methods to be tested ($MUTs$) and their respective unit test source code. The dataset obtained from ground-truth humans writing source code is denoted as $\mathcal{D}_h$. This is grouped by class and methods tests, as depicted in Eq. (2), where each $D_i \in \mathcal{D}_h$ comprises of:

(a) A parent class: Cls_i,
(b) The set of methods under test in this class: $MUT_j \in Cls_i$.
(c) The suite of unit tests corresponding for each MUT_j: $UT\left(MUT_j\right)_k$ (filtered, only those with status *passing* to ensure correctness of the selected data).

$$\mathcal{D}_h = \left(Cls_i,\ MUT_j,\ UT\left(MUT_j\right)_k\right)_i \tag{2}$$

3.2 Synthetic Dataset

The process of gathering supervised data, either from human annotators or via human feedback, as exemplified in [**?**], can be costly for programming tasks due to the necessity for professional developer involvement. In lieu of human feedback, we employ a method of execution feedback [2], to create artificial data containing unit-tests for fine-tuning the generative models.

Using the $\mathcal{D}_h$ dataset, the first version of fine-tuned unit-test code generation model is obtained, $GameUnitLLM_{v0}$. This is used further to generate correct synthetic unit-tests without human annotation, by passing this model to the function shown in Listing 1. First, a class and method are uniformly sampled, Lines 2, 3. Note that by using this strategy, new functions outside the initial set can be tested. The metadata of the class and function selected are passed to a prompt template then through the model to generate a new unit test U, Line 6 (detailed explanations are provided in Section 4.2). The resulted code is first compiled, Line 7. If it compiles, then a built is made and the correctness of the test is performed, by checking its *pass* status, Line 8. The last two steps are needed to ensure that if the test will be added to the dataset, then the model could learn clean data. Lastly, the coverage of the new U is compared to the coverage of the initial dataset, Line 9. If there is no improvement, either on the line or branch coverage (tested with the operator cov), then the test is discard. Otherwise, the new unit-test is added to the set, Line 10. The pipeline based process is also depicted in Figure 2.

```
1  GenerateUnitTest(LLM):
2  Sample a class Cls_i ∈ D
3  Sample a method to test MUT_j ∈ Cls_i
4  # Pass through the model the metadata for
5  # parent class and function to test
6  U = LLM(Prompt_gen(M(Cls_i), M(MUT_j)))
7  if compile(U):
8  if test_pass(execute(U)):
9  if cov(U) ⊄ cov(D_h):
10 D_s = D_s ∪ (Cls_i, MUT_j, U)
```

Listing 1: Steps for generating a synthetic dataset

3.3 Datset Numbers

The first step in the the dataset collection process as mentioned in Section 3 captured human written unit-tests collected from 3700 C++ and C# script files, 2941 classes, 319 MUTs and 726 corresponding unit tests. Out of these, 191 MUTs and 286 unit-tests were made for Unity and C# language. Furthermore, the split between training and evaluation is made by the 80/20 rule. The synthetic dataset was produced using the algorithm in Listing 1 by letting a server of 20 high-performance CPUs compiling and executing instances of unit-tests in a period of 7 day, 168h. While the compilation effort was not semnificative, execution time took between 1 second to 11 minutes. Table **??** shows how the number of tests that passes through each phase of the pipeline in this synthetic tests generation. The main observation at this preliminary step is that in general the model has more issues in producing valid C++ tests than using C#. Table 1 shows the final dataset grouped by the sizes by each split.

4 METHODS

4.1 Base Model

In this study, we utilize the Code Llama [22] models, which are built on the foundation of Llama 2. As of the time of writing, Llama 2 is regarded as the most sophisticated open-source model for code generation. Code Llama models are optimized for code generation, employing self-attention mechanisms to comprehend relationships and dependencies within the code. They are capable of generating both code and natural language descriptions of code from code and natural language prompts. They are compatible with a wide range of popular programming languages in use today, including Python, C++, Java, PHP, Typescript (Javascript), C#, and Bash.

Code Llama - Instruct represents a variant of Code Llama that has been fine-tuned and aligned with specific instructions. This process of instruction tuning extends the training phase, albeit with a distinct goal. The model is provided with an input in the form of a *natural language instruction* and the corresponding expected

[3] http://learn.unity.com
[4] https://www.unrealengine.com/marketplace/en-US/store

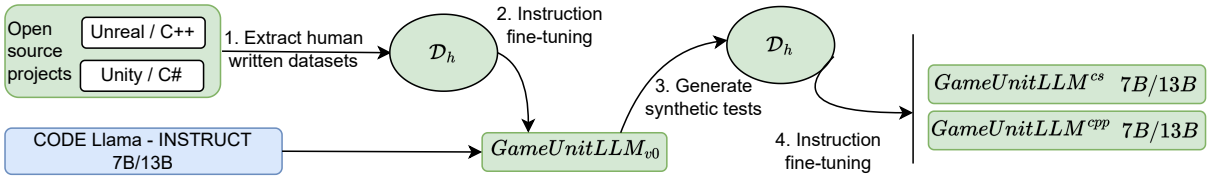

Figure 1: Refining the generative unit test model $GameUnitLLM$ begins with the base models of Code Llama [22] Instruct 7B represented by the blue rectangle. The fine-tuning process then proceeds with the extracted and processed datasets indicated by green rectangles). First, the already written human unit-tests considered as working are gathered, $\mathcal{D}_h$. Based on this, a fine-tuned version of generative unit-test model, $GameUnitLLM_{v0}$ is obtained. Infering this model, our method creates a synthetic dataset $\mathcal{D}_h$ without any human effort. Finally using the combined datasets, the same fine-tuning method produces the final generative models, one general independent of the programming language used, and one specific for C++ and $C\#$.

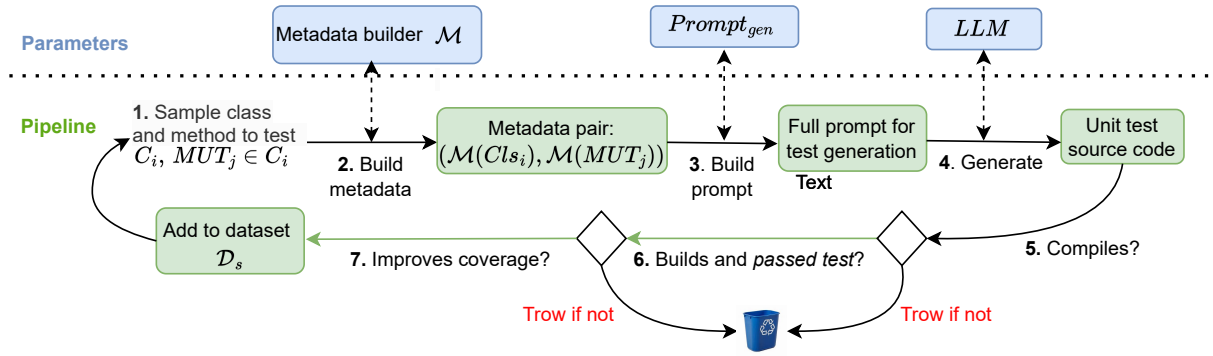

Figure 2: The pipeline for generating a batch of unit tests given the instances of different components as parameters. The main parameters (top row) are, from left to right: P1-the metada data builder given a class and and function, P2-the prompt template used to generate the test based on metadata, P3-the LLM instance used to take as input the prompt and generate one or more unit tests. The bottow row represents the continuous pipeline used to generate the unit tests, within 7 steps. The output produced by some individual steps is shown with green boxes. Generated tests are filtered before adding them to data synthetically generated datasets. The process checks if they compile, build as passed, and improve existing coverage.

Table 1: The size of the datasets splits needed further in evaluation. The columns contain the number of different methods under tests (MUTs), and the total of unit tests associated with these. The superscript *cs* is used for $C\#$, while *cpp* for $C++$. The subscript h represents the human written dataset, while s is the syntentic generated one.

Dataset	MUTs	Unit tests
D	590	1027
D^{cs}	384	434
D^{cpp}	206	593
D_{train}	472	822
D_{eval}	118	205
D_h	319	726
D_s	271	301
D_h^{cs}	191	268
D_h^{cpp}	128	458
D_s^{cs}	193	166
D_s^{cpp}	78	135

output. This enhances the model's ability to comprehend human expectations from their prompts.

The model comes with four released versions: 7B, 13B, 34B, and 70B (B represents billions of parameters). These address different latency or computational requirements. As an example, the 7B model could run reasonably on a single GPU or high-end CPU, being suitable for completing code in real time. The models can take up to 10^5 tokens of context (note that typical LLMs can take up 4096 tokens), which is an important feature from two points of view:

- Developers can provide more context from their source code. This is needed since the dataset $\mathcal{D}$'s, from our observations, have many classes that exceed 5×10^3 tokens.
- Debugging issues over large codebases. Usually, it is challenging for developers to stay on top of large code chunks, so having a model behind to understand and suggest or retrieve insights can alleviate the problem.

By using Code Llama as the base model, our suggested techniques can handle significantly larger input contexts and output tokens, i.e., up to $100k$ tokens. This is a substantial improvement over the studies in [24], and [25], which are constrained to a maximum of $4k$ tokens.

4.2 Refining the Base Model

We propose that fine-tuning is necessary to comprehend these back-end functions, their components, classes, methods, and interfaces.

The source code dataset $\mathcal{D}$, as defined in Eq. (2), undergoes additional processing through a series of steps outlined below. The objective is to automatically modify the source code instances and utilize them as an instruction refinement dataset for the base models.

The entire process from the base model to the refined version suggested by our techniques, *GameUnitGen*, is depicted in Figure 1. The refinement techniques and the source code in the repository are not tied to the Code Llama Instruct versions, meaning the techniques are compatible with model versions 7B, 13B, 34B, and 70B, where B represents billions of parameters.

Creating prompts and instructing LLM to generate tests. During both the training and testing phases, the LLM is instructed to generate tests using the prompt template shown in Listing 2. The primary concept is to design a reusable template with placeholder variables that can be populated as required during runtime. For instance, during the training phase, a dataset (explained below) is utilized to sample instances of inputs and versions of the anticipated outputs that the model should ideally generate. It's important to note that during the testing phase, to enhance user accessibility and in line with the methods proposed in [22], the user instructs the model to generate unit tests for a specific function and parent class using natural language. Automated scripts extract the necessary information from this request and populate the required placeholder variables in the prompt instance for user convenience.

During the training phase, the dataset $\mathcal{D}$, as defined in Eq. (2), which comprises source code instances of unit tests, is employed to construct an instruction refinement dataset, FTD. This is akin to the dataset utilized during the training of Code Llama [22]. By leveraging the reflection support in *C#*, automated scripts are devised to extract the necessary metadata for each available unit test source code. This extraction procedure identifies the context of the parent class, Cls_i, and the context of the method under test, i.e., its source code and signature, $MUT_j \in Cls_i$. This extraction process is denoted as $M_{i,j} = Metadata\left(Cls_i, MUT_j\right)$. This is subsequently used to populate the placeholder variables in the prompt outlined in Listing 2, i.e., $Prompt\left[M_{i,j}\right]$. The concept of populating templates using metadata from existing code bases is a general concept known as Retrieval Augmented Generation (RAG) [3].

From a compatibility standpoint, the utilization of native *C#* reflection support does not pose a limitation for its adoption in solutions outside of the Unity game engine. It's worth noting that reflection is currently managed by various methods within game engines that are based on programming languages that lack this native feature, such as C++ and the Unreal Engine[5], for instance.

$$\begin{aligned} M_{i,j} &= Metadata\left(Cls_i, MUT_j\right) \\ FTD_{i,j,k} &= \{Input: Prompt\left[M_{i,j}\right] \\ &\quad Answer: \left(MUT_j\right)_k\} \end{aligned} \tag{3}$$

Eq. 3 illustrates the technical structure of the dataset employed to refine the *GameUnitGen* model from the foundational Llama Code model. Each entry consists of a completed prompt, which serves as input to the model, and a corresponding ground truth test (deemed correct, i.e., *passing* evaluation) that the refined version is expected to produce. As indicated in Listing 3, the cross entropy [12] is utilized as a loss function to enhance the model's performance during the training phase by passing the input fields of entries sampled from the dataset FTD and comparing them with the corresponding (ground truth) test code for similarity. To assess the model's performance at the testing phase, the perplexity metric [16] is favored. FTD is further partitioned, akin to $\mathcal{D}$, into FTD_{train} and FTD_{test}.

```
Sample batch of examples B ~ FTD
# Pass through the model the inputs from B
# to obtained predicted answers
PredAnswers = GameUnitGen(B["Input"])
# Compute loss for optimization process by
    comparing
# how close is the output of the model
# compared to the example test.
Loss = CrossEntropy(PredAnswers, B["Answer"])
```

Listing 3: Fine-tuning process of the two GameUnitGen models.

Post-processing steps on the produced tests. Following the generation process, both during training and inference, the produced tests undergo post-processing to automatically enhance their accuracy before they are ultimately output. Specifically, each unit test is initially compiled with the parent class and the necessary libraries. At this stage, the majority of errors are due to syntax issues, such as: (a) test code including natural language explanations without comments, (b) duplication of the class/functions in the prompt, (c) minor alteration of imported packages, (d) generation of values for local variables beyond the range specified in their data type, (e) generation of incomplete mock classes. To address these commonly observed problems (a)-(e) as effectively as possible automatically, our approach devises a script capable of enhancing accuracy at the text level when the issue is basic and does not require human input.

5 EVALUATION

5.1 Details of Fine-tuning and Evaluation Setup

A fine-tuning procedure, as outlined in Section 4.2, is employed to modify the foundational Code Llama Instruct - 7B and 13B models using a dataset specific to a game environment. The outcome of this procedure is two models, each with the same number of parameters, which we refer to as GameUnitGen 7B and 13B respectively. Expanding the capacity with the larger model Code Llama Instruct - 70B could yield further enhancements but would require more computational resources. The choice to utilize models 7B and 13B was made to ensure that our methods and experiments could be executed on local systems, considering end-user GPUs for training on internal datasets, such as the NVidia RTX 4090 with 24 GB of video memory that we used for the experiments. Moreover, both models can also be utilized at test time on powerful CPUs like newer generation Intel Core i7-i9, which are capable of producing outputs with reasonable latency.

The fine-tuning method employed is the LoRA (Low-Rank Adaptation) [11], which can add updated weights to each selected layer of the base model with manageable computational costs. All layers were fine-tuned, with rank $r = 16$, $\alpha = 32$, $dropout = 0.05$. Default quantization parameters with $bfloat-16$ were used to optimize the training runtime performance. For the decoding process, we used nucleus sampling with $p = 0.95$ and $temperature = 0.8$. However,

[5] www.unrealengine.com

```
Generate {N} unit tests for function {FUNCTION_NAME} from class {PARENT_CLASS_NAME} considering\n\n:
### The target function signature is:  {FUNCTION_SIGNATURE}\n\n
### The target function corpus is: {FUNCTION_CORPUS} \n\n
### The parent class context code is: {PARENT_CLASS} \n\n
### The available mock classes interfaces are {MOCK_CLASSES} \n\n
### Target function local variables: {LOCAL_VARIABLES} \n\n
All generated tests should be enclosed using a <UnitGen></UnitGen> tag. Consider the following requirements\n:
1. Import required packages for types in {PARENT_CLASS_NAME}, {FUNCTION_SIGNATURE}, and {LOCAL_VARIABLES}.
2. Consider the mocks provided by input. If not available generate samples for each as required.
3. Add assert statements at the end of the test
```

Listing 2: The general template for the prompt used to create tests. The variables between curly brackets are placeholders that are filled at the time of training or testing

during the test time, the temperature was set to zero to achieve deterministic results that are predictable and repeatable, a decision inspired by previous research [24], [2]. The AdamW optimizer [15] is used, with $\beta 1 = 0.9$ and $\beta 2 = 0.95$, with a learning rate $\gamma = 3 \times 10^{-3}$, and a decay rate of 0.01. In the initial training phase, a warm-up of 200 steps was used.

The fine-tuning process was conducted over 5 epochs for both models and involved a total of 2055 steps (with a training data set size FTD_{train} of $\mathcal{D} = 411$ contextual unit tests). The literature indicates that the performance of the LLM training process does not increase as anticipated when iterating the same dataset and the same parameters [23], [14] multiple times, which was also observed in our case. Note that these parameters were determined empirically after brief training evaluations of different parameter ranges.

At the tools level, we used the Hugging Face[6] Transformers library and downloaded the base models publicly available on the platform. For the connection of interfaces and the prompt, we used the LangChain[7] APIs.

5.2 Quantitative Evaluation of Generated Tests

For each of the 268 MUTs present in the source code dataset, we instruct the model to generate 5 unit tests, leading to a total of 1340 produced unit tests. The experiments maintain the same quantity of generated tests and average the results acquired over 10 independent example processes.

Evaluation of Correctness. In accordance with prior research on test generation based on learning [18], [27], [29], [25], we assess the precision of the generated tests based on two primary criteria: (a) compilation precision, which checks if the test can be compiled successfully, and (b) execution precision, which determines if the test can pass execution (note that we are aware that each *MUT* is correct because it has been chosen to pass the tests). The comparative results between the two refined models and the three Code Llama-instruct base models are presented in Table 2, before and after implementing our automated post-generation corrections.

Based on our observation, the majority of the remaining compilation issues after the automated correction attempts were related to the language model hallucinations about symbols, functions (which do not exist in the API), incorrect number of parameters sent to functions, access to private members or instantiation of abstract classes. At the execution level, most of the issues were related to accessing invalid data and resources or calling APIs with functions that were not yet available at the time of testing.

Table 2: We conduct a comparative analysis between the three foundational CodeLlama Instruct models without any fine-tuning and our suggested models. The evaluation of the compilation's correctness is carried out after the model's initial output, following the automated processing attempts to rectify some prevalent issues, and after assessing the success of the execution post these modifications.

Model	Compilation (initial)	Compilation (processed)	Passing execution
CodeLlama 7B	26,64%	28,58%	18,28%
CodeLlama 13B	29,63%	32,61%	20,97%
CodeLlama 70B	33,43%	38,13%	22,46%
GameUnitGen7B	32,46%	38,73%	23,28%
GameUnitGen13B	37,39%	53,58%	28,58%

The results shown in Table 2 show two important observations:

- As anticipated, the fine-tuning process enhances precision in both compilation and execution. This can be intuitively understood as the base language model being refined with game engine-specific functionality, and its output tokens are then associated with these examples rather than general programming code.
- GameUnitGen 7B's performance surpasses that of Code Llama Instruct-70B, which at the time of testing necessitated approximately ten times the GPU computing power (i.e., 140 GB as opposed to 15 GB). Moreover, GameUnitGen 7B is compatible with CPU-only evaluation. On average, 5 unit tests with a filled contextual prompt are generated in less than 6 minutes on a Core i7-13700K. On the Nvidia RTX 4090 GPU, a similar experiment takes less than 2 minutes.

Assessment of Coverage. From this standpoint, we appraise the statement and branch coverage of the source code of the produced unit test over the source code of the MUTs. The outcomes in Table 3 juxtapose the two aforementioned metrics with the same models. This juxtaposition is employed to observe the overall enhancements of the generated tests compared to the original set of manually crafted unit tests in the extracted dataset. This adheres to the high-level approach utilized by Meta in [2] to generate and filter unit tests based on their utility.

Initially, the outcomes presented in Table 3 demonstrate the same advantage of the fine-tuning procedure as in the correctness assessment, i.e., considerably smaller and more computationally efficient

[6] https://huggingface.co/docs/transformers
[7] https://www.langchain.com

Table 3: Comparison between the three foundational models of CodeLlama Instruct without any fine-tuning and our suggested models from the viewpoint of coverage metrics. The average performance enhancement or reduction of the accurately generated tests in comparison to the examples in $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ are displayed.

Model	Branch coverage	Statement coverage
CodeLlama 7B	-21,2%	-31,58%
CodeLlama 13B	-14,6%	-27,14%
CodeLlama 70B	+14,2%	-0,13%
GameUnitGen 7B	+17,4%	+4,31%
GameUnitGen 13B	+29,3%	+9,58%

models can outperform in concentrated tasks or specific domains such as game development with a backend engine and APIs. In general, from both the correctness and coverage assessment viewpoints, we can deduce that the suggested techniques for automatically generating unit tests are not only beneficial but can also be executed on users' systems from a computational standpoint.

6 CONCLUSION

In this study, we explored the capabilities of LLMs in producing unit test code for the gaming industry. The prototype was developed using the Unity game engine, but the methods and source code provided can be modified to suit other game engines or general simulation software frameworks. From the experiments, it seems that the methods suggested in this study can enhance existing unit test sets or generate new ones from scratch, as the refined model can transfer its knowledge from one code base to another. A significant finding from our experiments is that fine-tuning on the specific use case of a game framework plays a substantial role in achieving the highest performance from an LLM. This is mainly because the model needs to understand the specific APIs of each project and how they are used in conjunction with the underlying structure and code of the game engine.

ACKNOWLEDGMENTS

This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS/CCCDI-UEFISCDI, project no. PN-IV-P8-8.1-PRE-HE-ORG-2023-0081, within PNCDI IV, and by the European Regional Development Fund, Competitiveness Operational Program 2014-2020 through project IDBC (code SMIS 2014+: 121512).

REFERENCES

[1] Bilal Al-Ahmad et al. 2021. Jacoco-coverage based statistical approach for ranking and selecting key classes in object-oriented software. *Journal of Engineering Science and Technology* 16 (08 2021), 3358–3386.
[2] Nadia Alshahwan et al. 2024. Automated Unit Test Improvement using Large Language Models at Meta. *32nd ACM Symposium on the Foundations of Software Engineering (FSE 24)* (2024).
[3] Ben Athiwaratkun et al. 2023. Multi-lingual Evaluation of Code Generation Models. arXiv:2210.14868 [cs.LG]
[4] Emily M. Bender et al. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT '21)*. Association for Computing Machinery, 610–623.
[5] José Campos et al. 2014. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Association for Computing Machinery, 55–66.
[6] Federico Cassano et al. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691.
[7] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]
[8] Arghavan Moradi Dakhel et al. 2023. GitHub Copilot AI pair programmer: Asset or Liability? arXiv:2206.15331 [cs.SE]
[9] Zhangyin Feng et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 1536–1547.
[10] Danielle Gonzalez et al. 2017. A Large-Scale Study on the Usage of Testing Patterns That Address Maintainability Attributes: Patterns for Ease of Modification, Diagnoses, and Comprehension. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 391–401.
[11] Edward J Hu et al. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=nZeVKeeFYf9
[12] Like Hui and Mikhail Belkin. 2020. Evaluation of neural architectures trained with square loss vs cross-entropy in classification tasks. *arXiv preprint arXiv:2006.07322* (2020).
[13] Raymond Li et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]
[14] Yiheng Liu et al. 2024. Understanding LLMs: A Comprehensive Overview from Training to Inference. arXiv:2401.02038 [cs.CL]
[15] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7
[16] Clara Meister and Ryan Cotterell. 2021. Language Model Evaluation Beyond Perplexity. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 5328–5339.
[17] Abhiman Neelakanteswara et al. 2024. RAGs to Style: Personalizing LLMs with Style Embeddings. In *Proceedings of the 1st Workshop on Personalization of Generative AI Systems*. Association for Computational Linguistics, St. Julians, Malta, 119–123. https://aclanthology.org/2024.personalize-1.11
[18] Pengyu Nie et al. 2023. Learning Deep Semantics for Test Completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2111–2123.
[19] Erik Nijkamp et al. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG]
[20] Hammond Pearce et al. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768.
[21] Cristiano Politowski et al. 2021. A Survey of Video Game Testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. 90–99. https://doi.org/10.1109/AST52587.2021.00018
[22] Baptiste Rozière et al. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]
[23] Noveen Sachdeva et al. 2024. How to Train Data-Efficient LLMs. arXiv:2402.09668 [cs.LG]
[24] Mohammed Latif Siddiq et al. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 71–82.
[25] Mohammed Latif Siddiq et al. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study.
[26] Philipp Straubinger and Gordon Fraser. 2023. A Survey on What Developers Think About Testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 80–90.
[27] Michele Tufano et al. 2021. arXiv. https://www.microsoft.com/en-us/research/publication/unit-test-case-generation-with-transformers-and-focal-context/
[28] Yue Wang et al. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.
[29] Zhiqiang Yuan et al. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv:2305.04207 [cs.SE]

Received 2024-03-28; accepted 2024-04-26