Forticode: A Benchmark for Evaluating the Robustness of Code Generation Models Against Adversarial Syntax Preserving Mutations

Stanford CS224N Custom Project

Amrit Baveja
Department of Computer Science
Stanford University
abaveja@stanford.edu

Anant Singhal
Department of Computer Science
Stanford University
saanant@stanford.edu

Abstract

As code generation models continue to grow in size, the likelihood of problems in benchmarks like HumanEval and MBPP being present in their training data increases. These benchmarks, which primarily consist of common computer science problems, may not provide a fair assessment of the models' capabilities. Additionally, pass@ scores, although somewhat useful for relative comparison, do not offer insights into the specific aspects of code generation where the models struggle. To address these limitations, we propose Forticode, a new benchmark that evaluates the performance of code generation models based on their robustness against adversarial syntax preserving mutations. We implemented nearly 50 syntax preserving AST transformations and instruction mutations that can be applied to HumanEval and MBPP code examples. By comparing the benchmark results of CodeLlama2 Instruct and Meta Llama3, we found that while CodeLlama2 generally performs worse in completing code with semantic mutations, it exhibits surprising resistance to instruction token replacement, variable obfuscation and renaming. This suggests that Codellama 2 may be better equipped to handle less readable code. Forticode provides a novel approach to assessing the robustness and generalization capabilities of code generation models, offering valuable insights beyond traditional evaluation metrics.

1 Key Information to include

- Mentor: Rashoon Poole
- External Collaborators (if you have any): Azalia Mirhoseini (SAIL)
- Sharing project: No

2 Introduction

In recent years, the development of large language models trained on source code has opened up new possibilities for automating and enhancing various aspects of software development. These advanced models can streamline time-consuming and repetitive tasks, such as debugging and generating documentation, by utilizing sophisticated algorithmic techniques (Barenkamp et al., 2020). They also enable structured analysis of large datasets, revealing hidden patterns and novel insights that can greatly improve the efficiency and effectiveness of software engineering practices. (Barenkamp et al., 2020) The impact of these models goes beyond routine tasks, as they have also shown promise in the area of test development (Battina, 2019). Empirical studies have demonstrated the transformative potential of these models. For example, one recent study found that developers who used an AI pair programming tool completed tasks 55.8% faster than those who did not use the tool (Battina, 2019).

The last few months have seen a wave of innovative code models from both commercial and open-source initiatives. OpenAI's Codex model, released a couple of years ago, set a notable benchmark in this field, showcasing the potential of large language models for source code. The evaluation of Codex used the HumanEval dataset (Chen et al., 2021), a carefully curated collection of 164 handcrafted programming problems designed to test a wide range of skills, such as language comprehension, logical reasoning, algorithmic proficiency, and basic math. Each problem in the HumanEval dataset includes a function signature, documentation, implementation body, and a comprehensive set of unit tests, with an average of 7.7 tests per problem. (Chen et al., 2021)

To measure the performance of code models on the HumanEval benchmark and similar datasets, the pass@k metric has become the standard. This metric represents the percentage of problems for which the model can generate a solution that successfully passes all associated unit tests within a specified number of attempts (k). (Chen et al., 2021) While the pass@k metric provides a high-level indication of a model's skill, it lacks the granularity needed to pinpoint the specific strengths and weaknesses of individual models. The binary nature of the metric—a problem is either solved or unsolved—doesn't capture the nuances in difficulty across different problems and the diverse set of skills required to solve them effectively. Moreover, the pass@k metric doesn't account for the potential memorization of solutions by the models, which can lead to inflated performance scores and hinder the accurate assessment of a model's true capabilities.

Recent advances in the field have introduced more sophisticated models that surpass the performance of Codex. Notable examples include DeepSeek Coder and CodeQwen 1.5, both of which have reported impressive results on the HumanEval and MBPP benchmarks.(DeepSeek-AI et al., 2024) (Bai et al., 2023) However, the evaluation of these models relies heavily on the pass@k metric, which, as discussed earlier, has limitations in terms of granularity and robustness to memorization.

The problem of memorization and data contamination poses a significant challenge in accurately assessing code models. While datasets like HumanEval try to mitigate this issue by using simple, handwritten problems, recent research has shown that code models still exhibit a concerning degree of memorization (Yang et al., 2024). A study conducted on a 1.5B parameter model found that a worrying 57% of the generated outputs contained memorized code snippets from the training data. This finding highlights the need for more robust evaluation methods that can effectively distinguish between genuine understanding and mere memorization. Another critical aspect of evaluating code models is the coverage of programming concepts within the benchmark datasets. A recent study Yadav and Singh (2024) analyzed the distribution of programming concepts in the HumanEval and MBPP datasets, revealing that five main concepts—Mathematics, Control Flow and Conditions, Basic Data Structures, Variables and Data Types, and Built-in Functions—account for over 70% of the problems. This imbalance and lack of comprehensive coverage raise concerns about the generalizability of the models' performance to real-world programming scenarios.

To address these limitations and provide a more comprehensive and robust evaluation framework for code models, we introduce Forticode. This novel benchmarking system uses adversarial mutations to enable a granular assessment of code models' capabilities while mitigating the impact of memorization. The key contributions of Forticode are:

- Introducing a benchmarking process that allows for a fine-grained analysis of a model's code completion abilities across various programming features. This process can be seamlessly integrated with existing code completion datasets, including HumanEval and MBPP.
- Developing a comprehensive set of nearly 50 rigorously tested abstract syntax tree mutations that preserve semantic equivalence, enabling the generation of diverse and challenging test cases.
- As far as we know, the first framework to apply adversarial robustness to avoid memorization in benchmarking language models.
- Creating an open-source framework that can be easily extended to accommodate the benchmarking of additional programming features and concepts, promoting collaboration and continuous improvement within the research community.

By using Forticode, we aim to provide a more accurate and nuanced understanding of the strengths and weaknesses of different code models. This granular evaluation approach will help researchers and practitioners make informed decisions when selecting and deploying code models in various software engineering contexts. Furthermore, the insights gained from Forticode will guide the development of future models, focusing on addressing the identified limitations and enhancing their overall performance and generalizability.

3 Related Work

TThe field of code modeling and evaluation has witnessed significant advancements in recent years, with researchers focusing on various aspects of model performance, robustness, and benchmarking. In this section, we discuss some of the most relevant and influential works that have shaped the current state of the art. Liu et al. Liu et al. (2023) introduced a novel approach to evaluate large language models for code generation by combining LLM-based and mutation testing techniques. Their methodology aimed to generate a more comprehensive set of test cases, addressing the limitations of existing benchmarks like HumanEval. Interestingly, their study revealed significant issues with the HumanEval dataset, including incomplete or incorrect test cases. By providing a corrected version of the dataset, they observed a 19.3% to 28.9% reduction in the pass@k metric across the models tested, highlighting the importance of rigorous evaluation and the impact of benchmark quality on model performance assessment. Yadav et al. Yadav and Singh (2024) proposed PythonSaga, a new benchmark designed to evaluate code-generating large language models. They crafted a set of 185 problems that cover a wide range of programming concepts, such as object-oriented programming, dynamic programming, and data structures. This approach aimed to mitigate the bias towards basic programming features found in previous benchmarks. While PythonSaga offers a more comprehensive representation of a model's performance compared to HumanEval, it has limitations. The human-written examples are time-consuming to create and may introduce bias. Moreover, establishing causality between specific code concepts and model performance remains challenging.

Li et al. (2022) explored the use of gradient-based adversarial attacks to assess the robustness of code classification models. Rather than using discrete and known mutations, they created a continuous and differentiable embeddings space to find candidates for gradient-based attacks that break the model's ability to classify the underlying source code. Their approach focused on renaming identifiers to avoid breaking semantic correctness. The authors found that incorporating a specific adversarial training process can help improve accuracy, but vanilla adversarial training alone is not sufficient to significantly reduce accuracy.

While these studies have made significant contributions to the field, they also expose the limitations of current benchmarking practices and the need for more comprehensive and granular evaluation frameworks. The reliance on human-written examples, the difficulty in establishing causality between specific code concepts and model performance, and the limited scope of code transformations used in robustness assessments are some of the key challenges that need to be addressed.

Forticode builds upon the insights gained from these previous works and aims to provide a more rigorous and comprehensive evaluation framework for code models. By leveraging adversarial mutations and a wide range of programming concepts, Forticode enables a granular assessment of model performance while mitigating the impact of memorization and data contamination. Our approach extends the scope of code transformations beyond identifier renaming, allowing for a more thorough evaluation of model robustness. Furthermore, the open-source nature of our framework facilitates collaboration and continuous improvement within the research community.

4 Approach

4.1 High-Level Overview

Let us assume we have a pretrained model M and our dataset D, which can be either HumanEval or MBPP. The following steps outline our approach:

1. **Seed Selection**: Prune D to find programming problems S, where $S \subseteq D$, with known solutions that maximize a seed selection criteria function.
2. **Canonical Solution Estimator**: For a given problem s, sample $k_c = 200$ solutions from the model and find the correct solution with the maximum cumulative log probability of the generated tokens. If no such solution can be found, remove that seed from consideration.
3. **Mutations**: Process the canonical solution and transform it into an AST. If the mutation operates directly on the source code, there is no need for AST transformation, and transformation sites are identified iteratively. If the mutation operates on the AST, walk the tree in depth-first order and detect valid transformation sites. Apply the mutation to each transformation site to produce mutated versions of the original source code.
4. **Stem Parsing**: Calculate the mutation point using the stem parsing algorithm and truncate the original and mutated code to the mutation point, yielding two prefixes.
5. **Sampling**: Sample $k_s = 200$ sequences from the model, prompting it to complete the original stem. Concatenate each sequence with its original stem to yield the full solution. Repeat the process with the mutated stem.
6. **Scoring**: Evaluate each sequence for correctness and calculate the pass@1, pass@5, and pass@10 scores for the original and mutated sequences. Provide the ratio of the pass@k scores between mutated and original, as well as the difference, to compare the model's performance relative to the baseline.

4.2 Seed Selection

Although HumanEval is a relatively small dataset ($n = 168$), each mutation can produce several new sequences that need to be tested, making the application of our benchmark to the entire dataset computationally expensive. To address this, we find k seeds that are likely to yield the most applicable mutations by computing a seed selection criteria function for each ground truth solution provided in the dataset. We have implemented the following metrics using the Radon library:

4.2.1 Cyclomatic Complexity

Cyclomatic complexity measures the number of linearly independent paths through a program's source code. McCabe (1976) It is calculated using the following formula:

$$CC = E - N + 2P \tag{1}$$

where E is the number of edges in the control flow graph, N is the number of nodes, and P is the number of connected components.

4.2.2 Halstead Volume

Halstead volume is a measure of the size of a program based on the number of operators and operands. Fenton and Pfleeger (2014) It is calculated using the following formula:

$$V = N \log_2 n \tag{2}$$

where N is the total number of operators and operands, and n is the number of unique operators and operands.

4.2.3 Logical Lines of Code

Logical lines of code (LLOC) is the number of executable statements in the source code, excluding comments and blank lines.
To find the seeds, we score each sequence and order them in descending order based on the seed criteria function. We then take the first k seeds.

4.3 Canonical Solution Estimator

To determine the types of mutations that prevent the model from completing the rest of the sequence, it is crucial to start from a sequence that the model can generate confidently. Given a seed, we sample $k_c = 200$ candidate sequences using only the prompt from the HumanEval dataset, with temperature $= 0.7$ and top-p $= 0.95$. We choose a temperature of 0.7 to encourage solution diversity in the canonical phase and use top-p $= 0.95$ following previous work Liu et al. (2023).

```
"""
Complete the rest of the below function such that it is self-contained
     and passes the corresponding tests. Write your code in a markdown
     code block, ending your response with ```. The function does not
    execute any tests of its logic. Don't include any testcases or
    evaluate your response.\n\n
"```python\n"
f"{stem.strip()}\n"
"```"
"""
```

Listing 1: Canonical Solution Estimation Prompt

We then score each candidate based on its correctness using the ground truth test cases from the EvalPlus dataset and execute the provided code in a security sandbox, similar to the approach used by OpenAI Liu et al. (2023). To improve processing speed, we run each solution on a separate CPU core via multiprocessing. The solutions are then cached to avoid recalculation in future runs. We filter down the correct solutions and find the solution with the maximum cumulative log probability, calculated using the following formula:

$$\log P(s) = \sum_{i=1}^{n} \log P(t_i | t_1, \ldots, t_{i-1}) \tag{3}$$

where s is the generated sequence, n is the number of tokens in the sequence, and t_i is the i-th token. We eliminate seeds that generate less than 10 out of 200 correct samples, as we are not confident in the model's ability to generate that code. The canonical solution is then postprocessed using Python's AST module to convert it into a syntax tree and back to code, ensuring consistent formatting with the outputs of the mutation transformers. This process adds parentheses around implicit tuples, normalizes strings to single quotes, and eliminates double line breaks.

4.4 Mutations

We apply each of our mutations to the canonical solution. The implemented mutations are categorized into code-based and comment/documentation-based transformations.

4.4.1 Code-based Mutations

A full list of all code-based mutations is in Appendix A. Most code-based transformations are implemented using the visitor design pattern. They are configured to look for certain types of nodes in the abstract syntax tree and replace them deterministically with new nodes.

```python
class ArrayToDictVisitor(OneByOneVisitor):

def is_transformable(self, node):
    return (
        isinstance(node, ast.Assign)
        and isinstance(node.value, ast.List)
        and len(node.value.elts) == 0
    )

def transform_node(self, node) -> list[ast.AST] | ast.AST:
    return ast.Assign(
        targets=node.targets, value=ast.Dict(keys=[], values=[],
        ↪  ctx=ast.Load())
```

```python
    )


class ArrayToDictTransformer(OneByOneTransformer, category=CRT.dicts):
    @property
    def visitor(self) -> Type[OneByOneVisitor]:
        return ArrayToDictVisitor

```

4.4.2 Comment/Documentation-based Mutations

Some mutations are string-based transformations, such as adding comments:

```python
class BlockCommentsTransformer(RegisteredTransformation,
↪   category=CRT.code_style):

    @property
    def comment(self):
        return "# I am a block comment\n# I am a block comment"

    @property
    def deterministic(self):
        return True  # Set to False to generate different variations

    def transform(self, code: str):
        new_lines = code.split("\n")
        results = []
        tree = asttokens.ASTTokens(code, parse=True).tree
        docstring_lines = get_docstring_ranges(tree)

        for i, line in enumerate(new_lines, start=1):
            if is_line_within_docstring(i, docstring_lines):
                continue
            if len(line.strip()) == 0:
                continue  # Skip empty lines
            indentation_level = len(line) - len(line.lstrip())
            if indentation_level > 0:
                copied = new_lines.copy()
                indented_comment = [
                    " " * indentation_level + cmt for cmt in
                    ↪  self.comment.split("\n")
                ]
                copied.insert(i - 1, "\n".join(indented_comment))
                results.append("\n".join(copied))

        return results

    @property
    def attack_func(self) -> callable:
        return self.transform
```

4.5 Docstring Mutations

While most of the mutations are trivial, it is worth to explain our process in determining docstring replacement candidates. To find semantically equivalent expressions of a given function's docstring d, we utilize WordNet, a lexical database for the English language that groups words into sets of synonyms called synsets. Given a function f with a docstring d, the process involves the following steps:

1. **Tokenization and Part-of-Speech Tagging**: Split d into individual words and identify their respective parts of speech using a POS tagger, such as the one provided by the NLTK library.
2. **Synonym Extraction**: For each word in d, query WordNet to obtain a list of synonyms that share the same part of speech.
3. **Replacement and Generation**: Replace each word in d with its synonyms iteratively to generate multiple semantically equivalent expressions. Ensure that replacements maintain grammatical consistency and meaning.
4. **Semantic Similarity Verification**: Compute the semantic similarity between the original and generated docstrings using metrics like cosine similarity on their word embeddings, obtained from models such as Word2Vec or BERT. This step ensures that the newly generated docstrings retain the original meaning.

Mathematically, if $d = (w_1, w_2, \dots, w_n)$ where w_i are words in the docstring, and $S(w_i)$ denotes the set of synonyms for w_i, the set of semantically equivalent docstrings D' can be defined as:

$$D' = \{(w'_1, w'_2, \dots, w'_n) \mid w'_i \in S(w_i) \cup \{w_i\}, \forall i \in [1, n]\}$$

Applying this to each candidate yields a list of mutations.

4.6 Choosing Mutatations

These mutations were chosen because each tests a specific aspect of code generation and understanding. Although not exhaustive, they provide granular detail as a starting point.

All implemented mutations are available in our GitHub repository. After applying a mutation, the solution is postprocessed to ensure stylistic parity, such as removing excessive newlines.

4.7 Stem Parsing

We attempt to find the first differing line between the original and mutated source code, with some caveats:

- Differing newlines are ignored.
- Mutations that generate multiple lines extend the mutated prefix to the end of their mutation. For example, if a for loop declaration is replaced with a counter initialization and a while statement, even though the first differing line is technically the counter, we extend the mutated prefix to the while statement.

To mimic the style a human would write in, we apply the popular Python "black" formatter to the source code, making it standardized.

4.8 Sampling

For the mutated and original prefixes, we use nucleus sampling to sample $k_s = 100$ candidate sample solutions to complete the code after the prefix. We use the following prompt:

~~~
"""
Complete the body of the below Python function such that it is self-
    contained and passes the corresponding tests. Write your code in a
     markdown code block, ending your response with ```. Don't include
     any testcases in your response.
```python
f"{definition.strip()}
```
"""
~~~

Listing 2: Pass@ Completion Prompt

We sample at temperature 0.5 following previous work, and use top-p $= 0.95$, as we did before. We use the VLLM library to perform batched inference using FlashAttention, which is more efficient.

4.9 Scoring

For both original and mutated sequences, we evaluate them against the EvalPlus ground truth, similar to the canonical solution analysis. We use the pass@k metric introduced in the OpenAI paper **?**:

$$\text{pass@k} := \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

where n is the number of problems, c is the number of correct solutions, and k is the argument.

We calculate the scores of the original sequence at pass@1, pass@5, and pass@10, as well as the mutated, and then calculate the ratios between the mutated and original scores, as well as their difference.

5 Experiments

This section contains the following.

Model Selection

For our experiments, we chose to compare the performance of CodeLlama2-7B-Instruct and Meta's Llama3 8B model. These models were selected for several reasons:

1. They represent different generations of the same base model, making for an interesting comparison.
2. CodeLlama2 has been specifically fine-tuned on code, allowing us to investigate whether this provides greater robustness against specific mutation styles compared to the general-purpose Llama3 model.

Dataset

We used the HumanEval dataset as our primary data source for the experiments. While we considered using the MBPP dataset, we found that it has less robust tests based on our own experiments. We also explored using EvalPlus's dataset, which offers more comprehensive tests. However, especially with MBPP, we observed that EvalPlus imposed tests on requirements that were not explicitly defined in the prompt, but may have been implied.

We used k=5 seeds because of computational constraints

Computational Resources

For our experiments, we utilized the following computational resources:

- 2x NVIDIA H100 GPUs for our VLLM server
- A machine with 64 Intel Xeon Platinum 8470 vCPUs and 128GB of RAM

Evaluation Metrics

We evaluated our models at temperature settings of 0.5 and calculated pass@1, pass@5, and pass@10 metrics. We hypothesized that higher pass@k scores should correspond with higher ratios across the board as pass@k only measures if one solution passes the test after k samples.

5.1 Results

The `pass@1` ratio scores were as follows, when averaged by mutation across candidates. We found that the same trends we saw in `pass@1` were present in `pass@5` and `pass@10`, as you can see in the Appendix. In this section, because of this, we will only discuss pass@1 scores.

5.2 Arrays

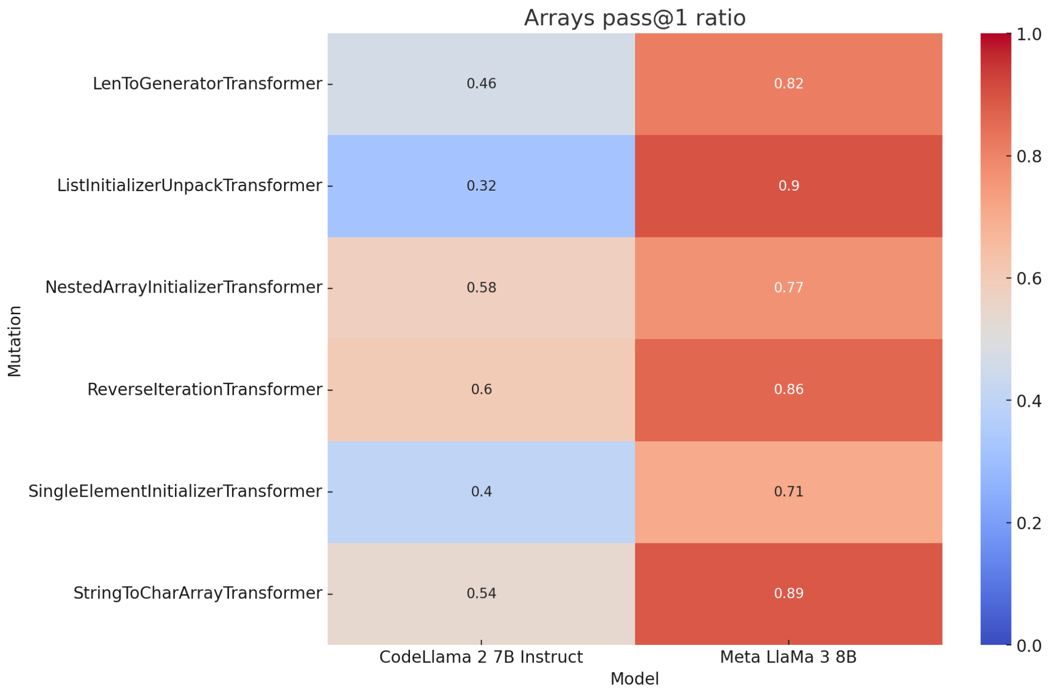

Figure 1: Array Mutations Pass@1 Ratio

5.3 Booleans

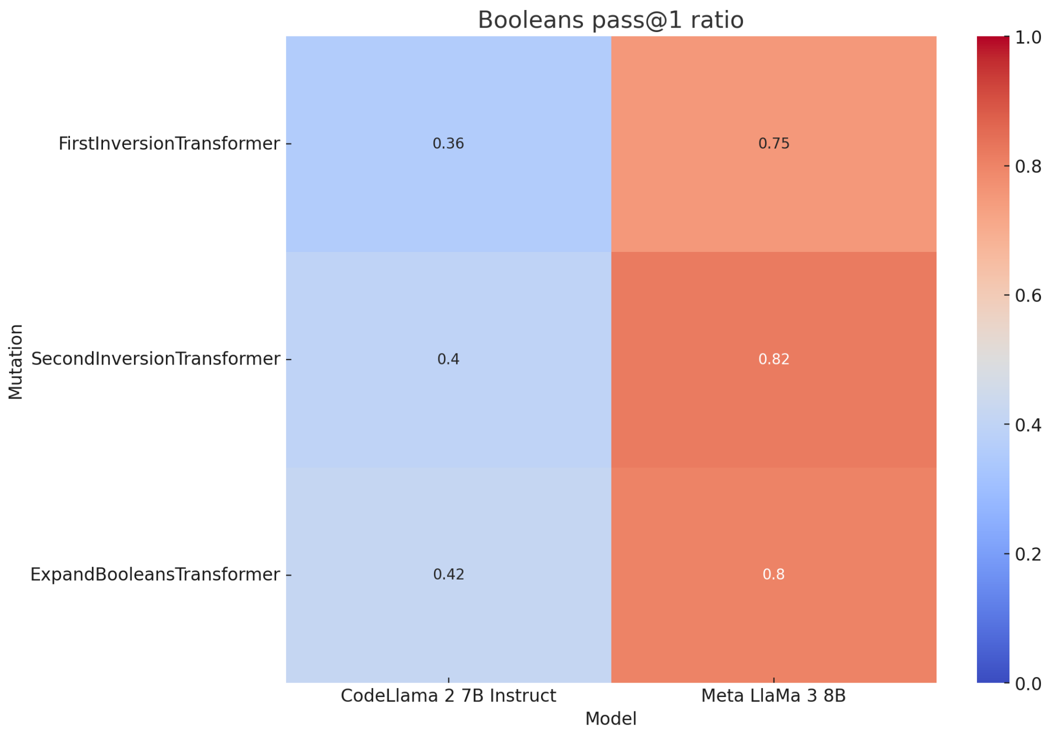

Figure 2: Boolean Mutations Pass@1 Ratio

5.4 Code Style

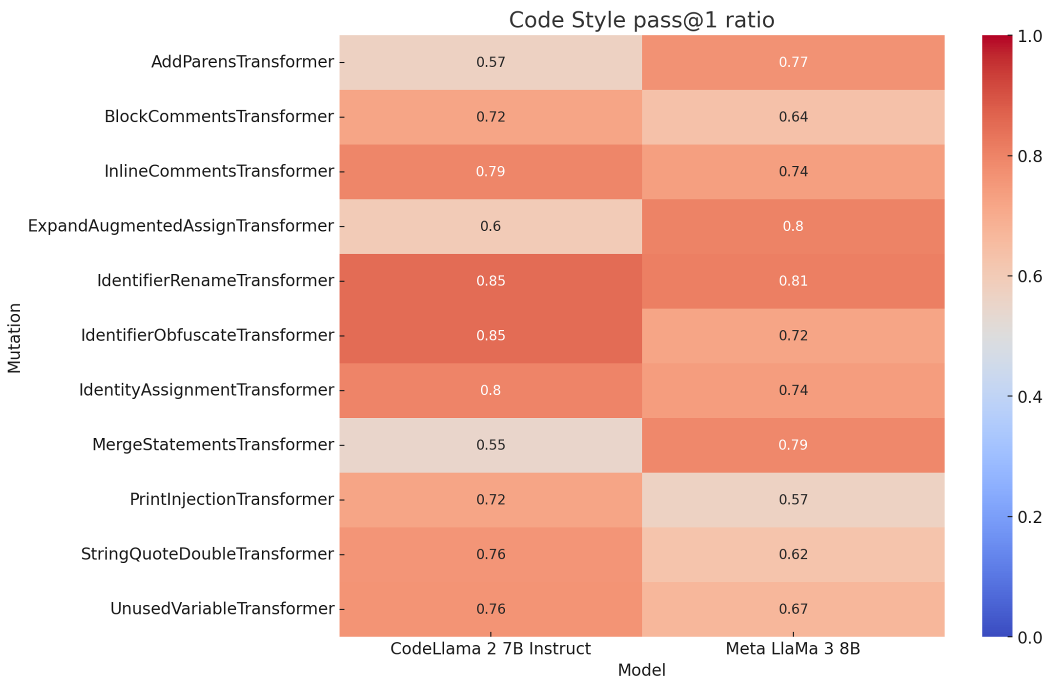

Figure 3: Code Style Pass@1 Mutation Ratios

5.5 Docstrings

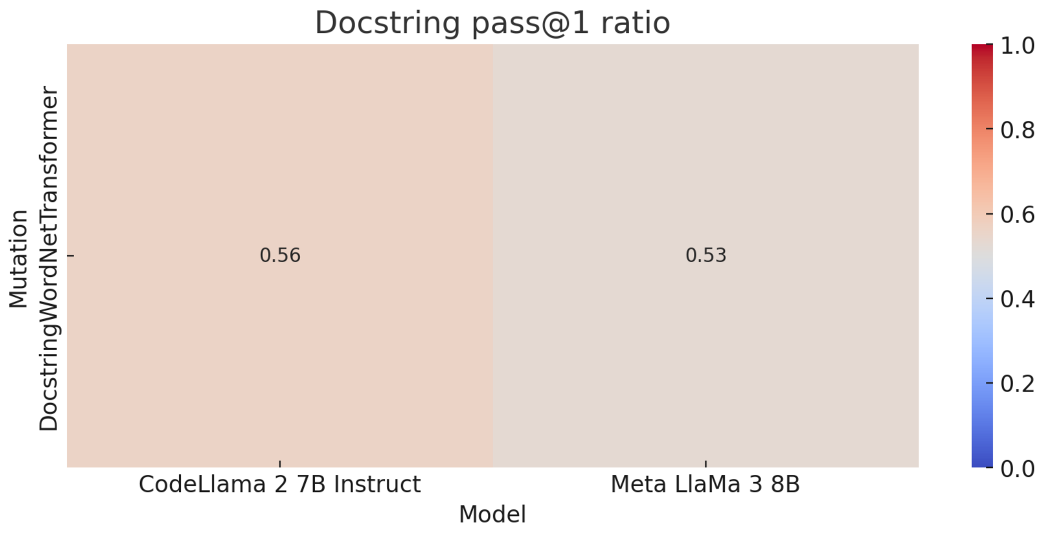

Figure 4: Docstring Pass@1 Mutation Ratios

5.6 Conditionals

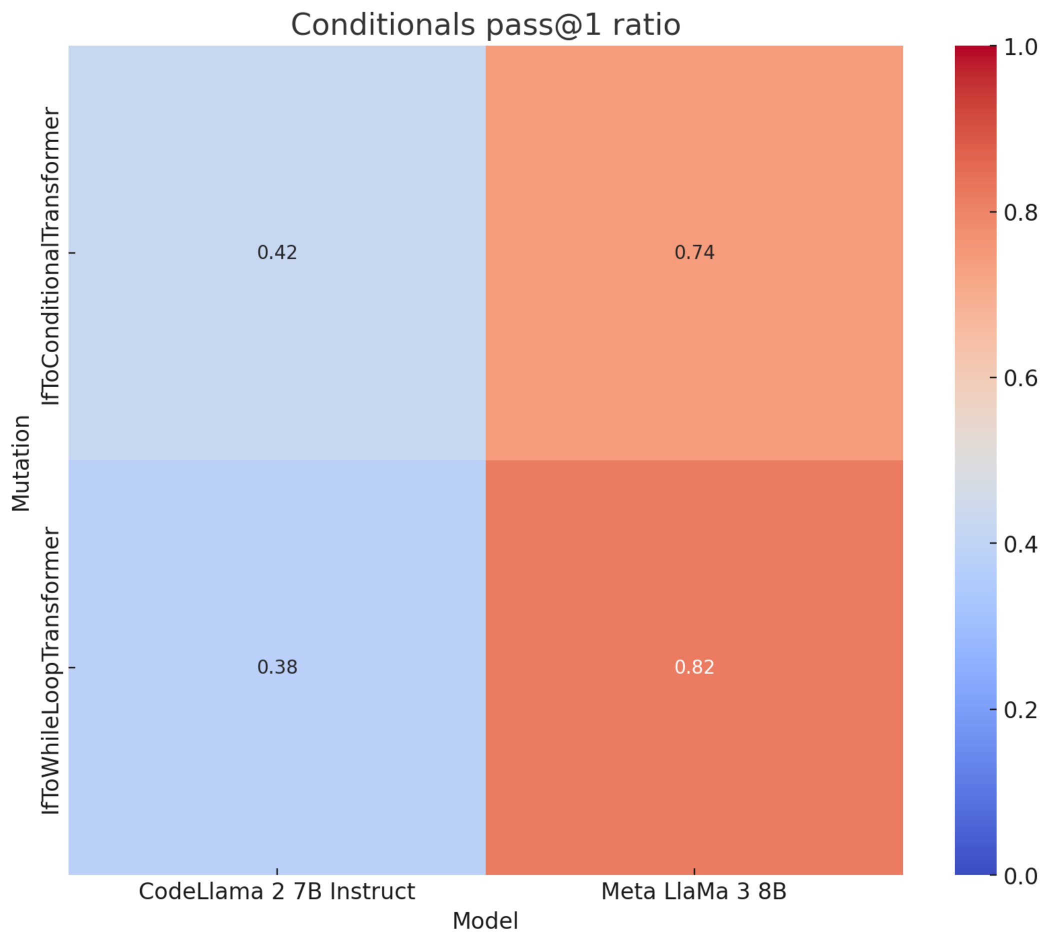

Figure 5: Conditional Pass@1 Mutation Ratios

5.7 Dictionaries

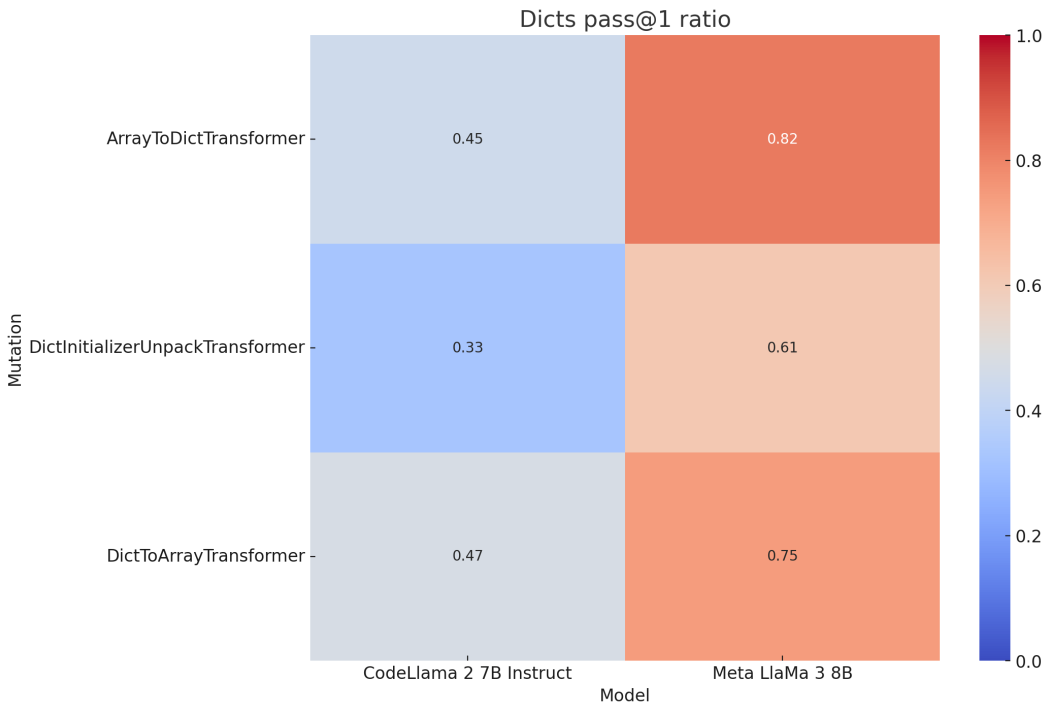

Figure 6: Dicts Pass@1 Mutation Ratios

5.8 Loops

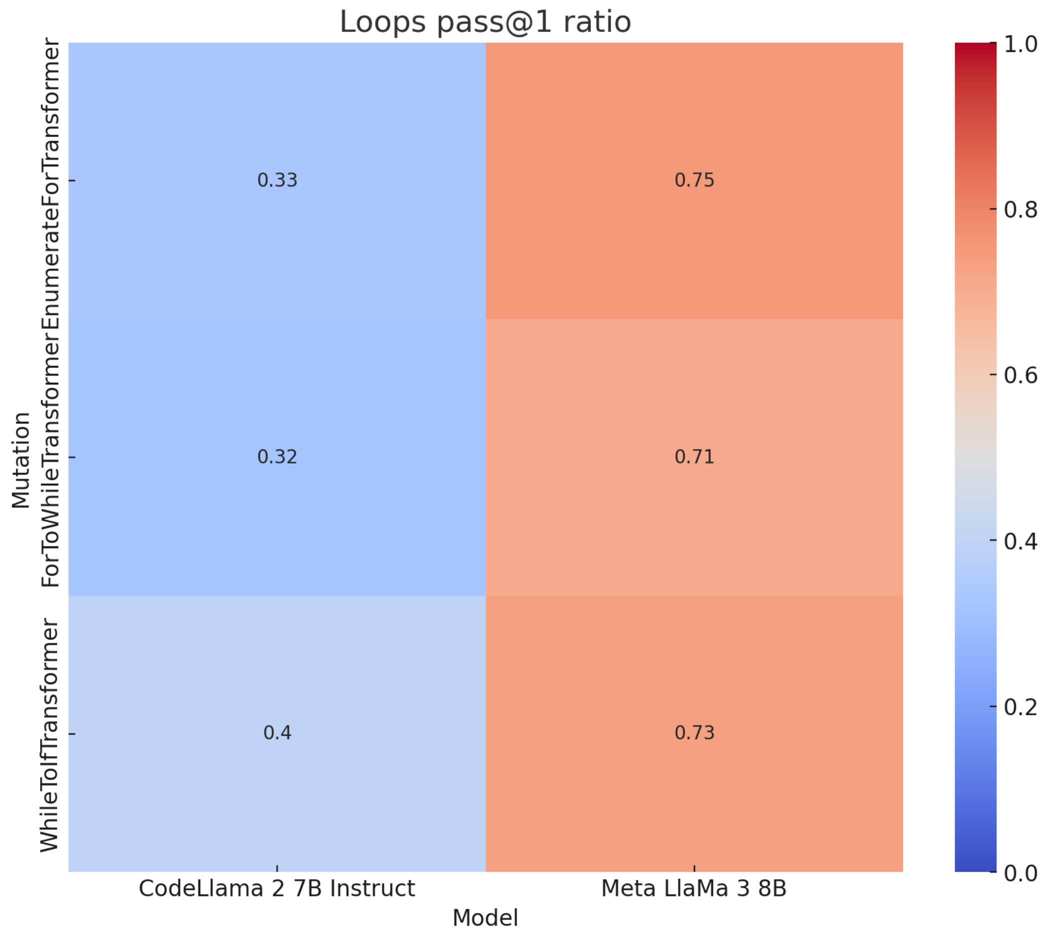

Figure 7: Loops Pass@1 Mutation Ratios

5.9 Math

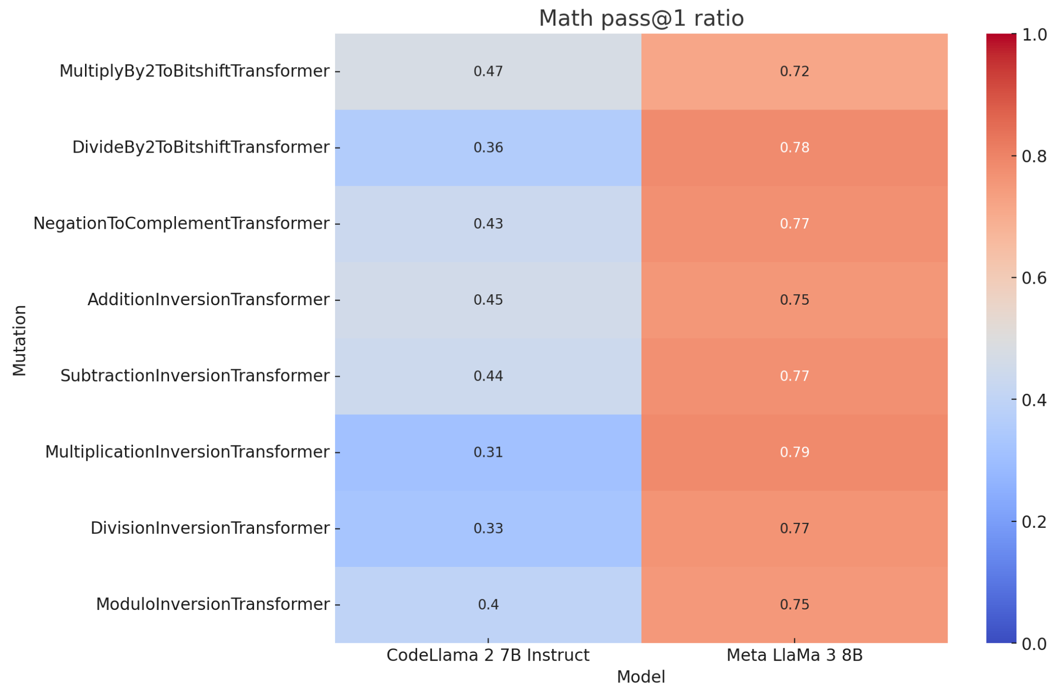

Figure 8: Math Pass@1 Mutation Ratios

5.10 Numbers

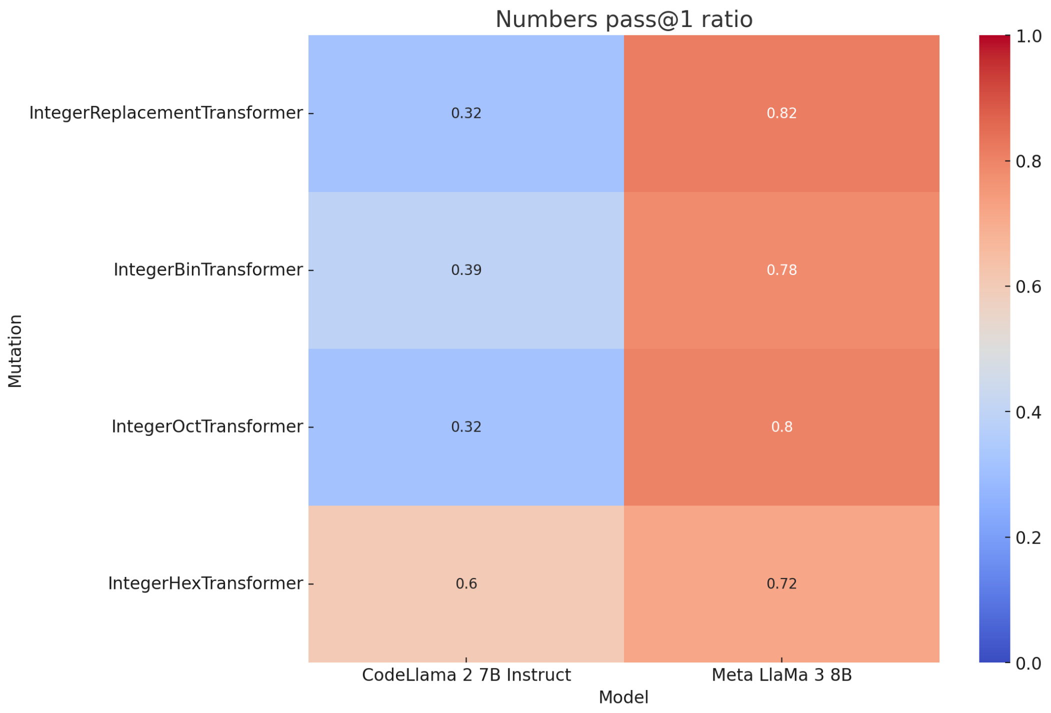

Figure 9: Numbers Pass@1 Mutation Ratios

5.11 Strings

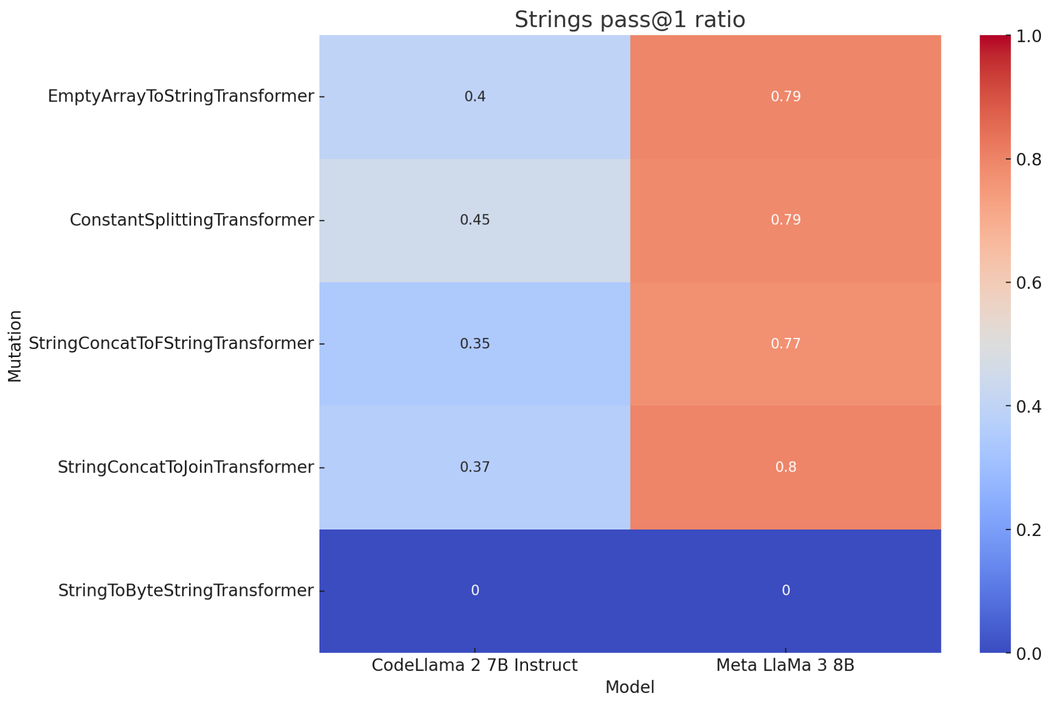

Figure 10: Strings Pass@1 Mutation Ratios

5.12 Summary

6 Analysis

Arrays Category

CodeLlama 2 7B Instruct has lower pass@1 ratios compared to Meta LlaMa 3 8B in all mutations. For example: LenToGeneratorTransformer (0.463 vs. 0.816).

Booleans Category

CodeLlama 2 7B Instruct has consistently lower pass@1 ratios. For example: FirstInversionTransformer (0.358 vs. 0.750).

Code Style Category

CodeLlama 2 7B Instruct shows higher pass@1 ratios in several mutations, particularly those involving stylistic changes and variable handling. - Higher scores for IdentifierRenameTransformer (0.850 vs. 0.811), IdentityAssignmentTransformer (0.800 vs. 0.741), and IdentifierObfuscateTransformer (0.850 vs. 0.721). For example: InlineCommentsTransformer (0.793 vs. 0.738).

Conditionals Category

Meta LlaMa 3 8B outperforms CodeLlama 2 7B Instruct. For example: IfToConditionalTransformer (0.425 vs. 0.744).

Dicts Category

Meta LlaMa 3 8B generally has higher pass@1 ratios. For example: DictInitializerUnpackTransformer (0.325 vs. 0.611).

Loops Category

Meta LlaMa 3 8B consistently outperforms CodeLlama 2 7B Instruct. For example: ForToWhileTransformer (0.321 vs. 0.709).

Math Category

Meta LlaMa 3 8B shows higher pass@1 ratios across all mutations. For example: AdditionInversionTransformer (0.454 vs. 0.754).

Numbers Category

Meta LlaMa 3 8B outperforms in most mutations, with notable differences. For example: IntegerReplacementTransformer (0.315 vs. 0.816).

Strings Category

Meta LlaMa 3 8B generally has higher pass@1 ratios. For example: StringConcatToFStringTransformer (0.347 vs. 0.770). Notable exception: StringToByteStringTransformer (both 0.000).

Docstring Category

CodeLlama 2 7B Instruct shows a slight edge with a pass@1 ratio of 0.56 compared to Meta LlaMa 3 8B's 0.53.

Suggested Model Competency

Meta LlaMa 3 8B Superiority

Meta LlaMa 3 8B generally outperforms CodeLlama 2 7B Instruct across most categories, particularly in Arrays, Booleans, Conditionals, Dicts, Loops, Math, and Numbers. This suggests that Meta LlaMa 3 8B has a higher competency in handling a variety of transformations, especially those involving complex data structures and logical operations.

Code Style Strengths for CodeLlama

In the Code Style category, CodeLlama 2 7B Instruct shows competitive or superior performance in specific mutations like BlockCommentsTransformer and InlineCommentsTransformer. It also has significantly higher scores in handling variable renaming and obfuscation, as shown by IdentifierRenameTransformer (0.850 vs. 0.811), IdentityAssignmentTransformer (0.800 vs. 0.741), and IdentifierObfuscateTransformer (0.850 vs. 0.721).

Overall Competency

The overall trend suggests that Meta LlaMa 3 8B is more robust and reliable across a wider range of transformations, highlighting its superior model competency. CodeLlama 2 7B Instruct, while having specific strengths in code style and variable handling, generally lags behind in terms of versatility and performance.

These findings suggest that while both models have their strengths, Meta LlaMa 3 8B is generally more competent and versatile, making it better suited for handling varied and obfuscated code. Its consistent performance across diverse categories indicates a higher robustness and reliability, highlighting its superiority in managing complex programming transformations compared to CodeLlama 2 7B Instruct.

Possible Reasons for Increased Performance in Code Style

The increased performance of CodeLlama 2 7B Instruct in the Code Style category might be attributed to its exposure to more diverse code samples, as it is specifically fine-tuned for code completion. According to Yang (2024), it is also possible that obfuscated or renamed code was present in the training data, which would suggest a higher probability of including those tokens in the output. Yang et al. (2024) Further analysis is needed to understand this more in-depth, but this provides a good starting point for understanding the strengths of CodeLlama 2 7B Instruct in handling stylistic and obfuscated code transformations.

7 Conclusion

In this paper, we introduced Forticode, a novel benchmarking framework that evaluates the robustness and generalization capabilities of code generation models against adversarial syntax-preserving mutations. By implementing nearly 50 AST transformations and instruction mutations, Forticode enables a granular assessment of model performance across various programming concepts and features. This approach addresses the limitations of existing benchmarks, such as HumanEval and MBPP, which may be susceptible to data contamination and lack comprehensive coverage of programming concepts. Our experiments, comparing the performance of CodeLlama2 Instruct and Meta Llama3, revealed interesting insights into the strengths and weaknesses of these models. While Meta Llama3 generally outperformed CodeLlama2 in completing code with semantic mutations, CodeLlama2 exhibited surprising resistance to instruction token replacement, variable obfuscation, and renaming. This suggests that CodeLlama2 may be better equipped to handle less readable code, possibly due to its exposure to more diverse code samples during fine-tuning. The main achievements of our work include:

1. Developing a comprehensive set of rigorously tested AST mutations that preserve semantic equivalence, enabling the generation of diverse and challenging test cases.
2. Providing a granular evaluation approach that offers valuable insights beyond traditional metrics like pass@k scores.
3. Demonstrating the effectiveness of Forticode in identifying specific strengths and weaknesses of code generation models, as exemplified by the comparison between CodeLlama2 and Meta Llama3.

However, our work also has some limitations:

1. The current set of mutations, while diverse, may not cover all possible programming concepts and features exhaustively.
2. The computational resources required to apply Forticode to large datasets can be substantial, limiting the number of seeds that can be practically evaluated.
3. The insights gained from Forticode are based on a comparison of two specific models, and further research is needed to generalize these findings to other code generation models.

Future work should focus on:

1. Expanding the set of mutations to cover a wider range of programming concepts and features, further enhancing the comprehensiveness of the evaluation.
2. Optimizing the computational efficiency of Forticode to enable its application to larger datasets and more diverse code generation models.
3. Investigating the relationship between model architecture, training data, and performance on specific mutation categories to gain a deeper understanding of the factors that contribute to model robustness and generalization.
4. Exploring the potential of using insights from Forticode to guide the development of more robust and generalizable code generation models.

In conclusion, Forticode represents a significant step forward in the evaluation of code generation models, providing a novel approach to assess their robustness and generalization capabilities. By offering granular insights into model performance across various programming concepts and features, Forticode complements existing benchmarks and contributes to a more comprehensive understanding of the strengths and limitations of code generation models. As the field continues to evolve, we believe that Forticode will play an important role in guiding the development of more advanced and reliable code generation models.

8 Ethics Statement

8.0.1 Misuse and Unintended Consequences

- Forticode's ability to generate a wide range of mutations and test cases could potentially be misused by bad actors to find vulnerabilities in code and exploit them for malicious purposes. For example, attackers could use Forticode to automatically generate adversarial examples that bypass security checks or exploit weaknesses in software systems. Moreover, the granular insights gained on model capabilities could be used to develop more sophisticated attacks or to target specific systems more effectively.
- The increased model robustness resulting from Forticode's testing could also be used to generate more convincing code that perpetuates misinformation, disinformation, scams, and spam. If the language models become better at generating coherent and persuasive text, they may be employed to create fake news articles, phishing emails, or other forms of deceptive content that are more difficult to detect and combat.
- **Mitigation**: To mitigate these risks, it is crucial to put appropriate access controls and usage restrictions in place for Forticode. This may involve implementing authentication and authorization mechanisms to ensure only authorized users can access the tool and its results. Additionally, usage monitoring and logging should be employed to detect and prevent misuse. It is also important to maintain accountability and transparency on how Forticode is being used and shared, with clear guidelines and policies governing its appropriate use.

8.0.2 Privacy and Security

- The code repositories used to train and test the models in Forticode may contain sensitive data or private information that could be inadvertently leaked or exposed. For example, if the code contains hard-coded API keys, passwords, or other credentials, they may be included in the generated mutations and test cases, potentially compromising the security of the associated systems or services. Similarly, if the code contains personal information such as names, addresses, or social security numbers, this data could be unintentionally disclosed through the generated outputs.
- Forticode's automated mutation and testing capabilities could also be used to find and exploit security vulnerabilities in the code itself. By systematically generating a wide range of test cases and probing the code for weaknesses, attackers could use Forticode to discover and exploit vulnerabilities such as buffer overflows, injection attacks, or memory corruption issues. This could compromise the integrity and confidentiality of the systems running the code, as well as any associated data or resources.
- **Mitigation**: To address these privacy and security risks, it is essential to ensure that all code and data used in Forticode is properly sanitized and anonymized. This involves removing any sensitive information such as credentials, keys, or personal data from the code before using it for training or testing. Additionally, strict data governance policies should be put in place to ensure that all data is handled securely and in compliance with relevant regulations such as GDPR or HIPAA. Access to the code and data should be restricted to authorized personnel only, and all outputs generated by Forticode should be carefully reviewed for any potential leaks or disclosures before being shared or published.

References

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng

Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Marco Barenkamp, Jonas Rebstadt, and Oliver Thomas. 2020. Applications of ai in classical software engineering. *AI Perspectives*, 2.

Dhaya Sindhu Battina. 2019. Artificial intelligence in software test automation: A systematic literature review.

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

DeepSeek-AI, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Li Y K, Wenfeng Liang, Fangyun Lin, Liu A X, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, Xu R X, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv (Cornell University)*.

Norman E. Fenton and Shari L. Pfleeger. 2014. An analysis of the design and definitions of halstead's metrics. *ResearchGate*.

Yiyang Li, Hongqiu Wu, and Hai Zhao. 2022. Semantic-preserving adversarial code comprehension.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation.

Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Ankit Yadav and Mayank Singh. 2024. Pythonsaga: Redefining the benchmark to evaluate code generating llm.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2024. Unveiling memorization in code models.

A Appendix

A.1 Transformation Functions

Table 1: Mutation Pass Ratios

Mutation Name	Category	Functionality
LenToGeneratorTransformer	arrays	Converts len() calls to sum([1 for _ in arr]) expressions
ListInitializerUnpackTransformer	arrays	Replaces list initialization with list(*[...])
NestedArrayInitializerTransformer	arrays	Converts single list initialization to nested list initialization
ReverseIterationTransformer	arrays	Reverse iteration order in for loops
SingleElementInitializerTransformer	arrays	Convert an empty array initializer to [None]
StringToCharArrayTransformer	arrays	Converts strings to character arrays
FirstInversionTransformer	booleans	Inverts boolean expression, applying DeMorgan's law
SecondInversionTransformer	booleans	Double inverts a boolean expression
ExpandBooleansTransformer	booleans	Expand a boolean to a compound expr a $\rightarrow$ (a and True)
AddParensTransformer	code style	Adds parentheses around expressions
ExpandAugmentedAssignTransformer	code style	Expands augmented assignments into full assignments
IdentifierRenameTransformer	code style	Renames identifiers to single letter names
IdentifierObfuscateTransformer	code style	Obfuscates identifiers with random 8 character strings
IdentityAssignmentTransformer	code style	Introduces identity assignments (e.g., x = x)
MergeStatementsTransformer	code style	Merges multiple statements into a single line
PrintInjectionTransformer	code style	Injects print statements into the code
StringQuoteDoubleTransformer	code style	Converts single-quoted strings to double-quoted strings
UnusedVariableTransformer	code style	Introduces unused variable declarations
IfToConditionalTransformer	conditionals	Converts compatible if statements to conditional expressions
IfToWhileLoopTransformer	conditionals	Converts if statements to while loops
ArrayToDictTransformer	dicts	Converts arrays to dictionaries
DictInitializerUnpackTransformer	dicts	a = {} becomes a = dict(**{})
DictToArrayTransformer	dicts	Converts dictionaries to arrays
EnumerateForTransformer	loops	Introduces enumerate() calls in for loops
ForToWhileTransformer	loops	Converts for loops to while loops
WhileToIfTransformer	loops	Converts while loops to if statements
MultiplyBy2ToBitshiftTransformer	math	Converts multiplications by 2 to bitshift operations
DivideBy2ToBitshiftTransformer	math	Converts divisions by 2 to bitshift operations
NegationToComplementTransformer	math	Converts negation to bitwise complement
AdditionInversionTransformer	math	Inverts addition operations
SubtractionInversionTransformer	math	Inverts subtraction operations
MultiplicationInversionTransformer	math	Inverts multiplication operations
DivisionInversionTransformer	math	Inverts division operations
ModuloInversionTransformer	math	Inverts modulo operations
IntegerReplacementTransformer	numbers	Replaces integers with equivalent expressions
IntegerBinTransformer	numbers	Converts integers to binary notation
IntegerOctTransformer	numbers	Converts integers to octal notation
IntegerHexTransformer	numbers	Converts integers to hexadecimal notation
EmptyArrayToStringTransformer	strings	Converts empty arrays to empty strings
ConstantSplittingTransformer	strings	Splits string constants into substrings
StringConcatToFStringTransformer	strings	Converts string concatenation to f-string interpolation
StringConcatToJoinTransformer	strings	Converts string concatenation to join() calls
StringToByteStringTransformer	strings	Converts strings to byte strings

A.2 All Results

Table 2: Mutation Pass Ratios

Mutation	Metric	CodeLlama 2 7B Instruct	Meta LlaMa 3 8B
LenToGeneratorTransformer	pass@1	0.463002	0.815750
	pass@5	0.696139	0.950000
	pass@10	0.776494	0.980000
ListInitializerUnpackTransformer	pass@1	0.318823	0.896864

Continued on next page

Table 2 – continued from previous page

Mutation	Metric	CodeLlama 2 7B Instruct	Meta LlaMa 3 8B
	pass@5	0.557604	0.950000
	pass@10	0.645718	0.980000
NestedArrayInitializerTransformer	pass@1	0.579679	0.765559
	pass@5	0.826145	0.883834
	pass@10	0.911791	0.916687
ReverseIterationTransformer	pass@1	0.597610	0.861406
	pass@5	0.805207	0.950000
	pass@10	0.919035	0.980000
SingleElementInitializerTransformer	pass@1	0.398883	0.706408
	pass@5	0.580795	0.900000
	pass@10	0.674297	0.980000
StringToCharArrayTransformer	pass@1	0.537843	0.889764
	pass@5	0.787122	0.950000
	pass@10	0.874049	0.980000
FirstInversionTransformer	pass@1	0.358297	0.749834
	pass@5	0.580877	0.900000
	pass@10	0.673613	0.980000
SecondInversionTransformer	pass@1	0.398049	0.817537
	pass@5	0.582712	0.900000
	pass@10	0.672912	0.980000
ExpandBooleansTransformer	pass@1	0.417219	0.802870
	pass@5	0.627506	0.950000
	pass@10	0.722620	0.980000
AddParensTransformer	pass@1	0.573407	0.769476
	pass@5	0.793396	0.950000
	pass@10	0.894053	0.980000
BlockCommentsTransformer	pass@1	0.720839	0.636216
	pass@5	0.848325	0.847243
	pass@10	0.940059	0.947212
InlineCommentsTransformer	pass@1	0.793305	0.737544
	pass@5	0.863554	0.837897
	pass@10	0.937669	0.931265
ExpandAugmentedAssignTransformer	pass@1	0.418925	0.802205
	pass@5	0.620328	0.900000
	pass@10	0.725022	0.980000
IdentifierRenameTransformer	pass@1	0.475012	0.811261
	pass@5	0.698282	0.950000
	pass@10	0.791579	0.980000
IdentifierObfuscateTransformer	pass@1	0.510324	0.720940
	pass@5	0.725154	0.877631
	pass@10	0.802005	0.943737
IdentityAssignmentTransformer	pass@1	0.368114	0.740871
	pass@5	0.613485	0.900000
	pass@10	0.708582	0.980000
MergeStatementsTransformer	pass@1	0.448034	0.791452
	pass@5	0.682343	0.900000
	pass@10	0.774261	0.980000
PrintInjectionTransformer	pass@1	0.719938	0.569037
	pass@5	0.831939	0.778628
	pass@10	0.926537	0.887008
StringQuoteDoubleTransformer	pass@1	0.758346	0.624812
	pass@5	0.839112	0.835128
	pass@10	0.919042	0.938992
UnusedVariableTransformer	pass@1	0.762387	0.668642
	pass@5	0.837553	0.812921

Continued on next page

Table 2 – continued from previous page

Mutation	**Metric**	**CodeLlama 2 7B Instruct**	**Meta LlaMa 3 8B**
	pass@10	0.923693	0.928754
IfToConditionalTransformer	pass@1	0.425346	0.744170
	pass@5	0.660239	0.900000
	pass@10	0.758174	0.980000
IfToWhileLoopTransformer	pass@1	0.380125	0.819217
	pass@5	0.608761	0.900000
	pass@10	0.700978	0.980000
ArrayToDictTransformer	pass@1	0.448492	0.824365
	pass@5	0.689496	0.900000
	pass@10	0.789434	0.980000
DictInitializerUnpackTransformer	pass@1	0.324872	0.610578
	pass@5	0.570019	0.900000
	pass@10	0.693095	0.980000
DictToArrayTransformer	pass@1	0.473253	0.746404
	pass@5	0.691870	0.900000
	pass@10	0.793423	0.980000
EnumerateForTransformer	pass@1	0.329294	0.749376
	pass@5	0.563670	0.900000
	pass@10	0.675308	0.980000
ForToWhileTransformer	pass@1	0.321197	0.708690
	pass@5	0.568611	0.900000
	pass@10	0.674702	0.980000
WhileToIfTransformer	pass@1	0.398474	0.732273
	pass@5	0.597639	0.900000
	pass@10	0.713994	0.980000
MultiplyBy2ToBitshiftTransformer	pass@1	0.474007	0.718046
	pass@5	0.683410	0.900000
	pass@10	0.771696	0.980000
DivideBy2ToBitshiftTransformer	pass@1	0.358986	0.782953
	pass@5	0.569318	0.900000
	pass@10	0.675036	0.980000
NegationToComplementTransformer	pass@1	0.435365	0.770768
	pass@5	0.649989	0.900000
	pass@10	0.743598	0.980000
AdditionInversionTransformer	pass@1	0.453682	0.753644
	pass@5	0.686244	0.900000
	pass@10	0.772542	0.980000
SubtractionInversionTransformer	pass@1	0.439364	0.773442
	pass@5	0.666324	0.900000
	pass@10	0.766285	0.980000
MultiplicationInversionTransformer	pass@1	0.312128	0.786820
	pass@5	0.552355	0.900000
	pass@10	0.654658	0.980000
DivisionInversionTransformer	pass@1	0.326456	0.764808
	pass@5	0.551496	0.900000
	pass@10	0.667197	0.980000
ModuloInversionTransformer	pass@1	0.397847	0.750230
	pass@5	0.629506	0.900000
	pass@10	0.727091	0.980000
IntegerReplacementTransformer	pass@1	0.315303	0.816378
	pass@5	0.552924	0.900000
	pass@10	0.654885	0.980000
IntegerBinTransformer	pass@1	0.393742	0.782623
	pass@5	0.597757	0.900000
	pass@10	0.703256	0.980000

Continued on next page

Table 2 – continued from previous page

Mutation	Metric	CodeLlama 2 7B Instruct	Meta LlaMa 3 8B
IntegerOctTransformer	pass@1	0.315425	0.804204
	pass@5	0.548350	0.900000
	pass@10	0.645568	0.980000
IntegerHexTransformer	pass@1	0.604885	0.718066
	pass@5	0.824935	0.900000
	pass@10	0.919144	0.980000
EmptyArrayToStringTransformer	pass@1	0.398353	0.794083
	pass@5	0.594417	0.900000
	pass@10	0.708748	0.980000
ConstantSplittingTransformer	pass@1	0.453009	0.788524
	pass@5	0.680504	0.900000
	pass@10	0.765569	0.980000
StringConcatToFStringTransformer	pass@1	0.347119	0.770496
	pass@5	0.588773	0.900000
	pass@10	0.688074	0.980000
StringConcatToJoinTransformer	pass@1	0.368002	0.797971
	pass@5	0.581564	0.900000
	pass@10	0.699295	0.980000
StringToByteStringTransformer	pass@1	0.000000	0.000000
	pass@5	0.000000	0.000000
	pass@10	0.000000	0.000000