Otto von Guericke University Magdeburg

Faculty of Computer Science

Development and Application of a Robustness Evaluation Framework for LLM-based C-to-Rust Translations

Master Thesis

Author:

Martin Weiss

Supervisors:

Prof. Dr. Andreas Schmietendorf
Otto von Guericke University Magdeburg

Dr. Jesko Hecking-Harbusch
Robert Bosch GmbH

Magdeburg, 15.05.2025

Weiss, Martin:
Development and Application of a Robustness Evaluation Framework for LLM-based C-to-Rust Translations
Master Thesis, Otto von Guericke-University Magdeburg, 2025.

Abstract

Legacy codebases written in *C* have been shown to fall victim to memory safety vulnerabilities, making a migration into safer languages like *Rust* highly desirable. Manually translating the vast amounts of code is infeasible, and traditional rule-based transpilers fail to produce *idiomatic* and *safe Rust* code. The coding capabilities of Large Language Models (LLMs) present a promising alternative for an *idiomatic* translation. However, their practical usability requires a comprehensive investigation, especially regarding their robustness when translating the diverse nature of real-world *C* codebases. The thesis addresses this need by developing and applying a comprehensive robustness evaluation framework tailored to LLM-based *C-to-Rust* translation.

This framework evaluates the LLMs' susceptibility to prompt variations through a diverse set of code-focused perturbations. It utilizes an existing state-of-the-art code translation system that directly checks for *functional equivalence* and performs automatic *feedback loops* in case of failure. Quantifying translation performance deviations to perturbed and unperturbed inputs, the framework's metrics allow a detailed view of the robustness. The framework complements this information by including whether perturbation-based performance deviations go beyond the expected fluctuations of LLMs, whether *feedback loops* impact robustness, and whether semantic similarity between input variations can predict the robustness of LLMs.

Applying the framework reveals that modern models are robust against most input variations, especially when supported by *feedback loops*. The *feedback loops* improved both the overall translation success and the robustness, making them an indispensable component of LLM-based translation systems. However, the evaluation still discovered weaknesses to certain perturbations, one model-agnostic and multiple model-specific. While modern models predominantly signal robust behavior, they exhibit unique strengths and weaknesses that impact the choice of the best model for the translation system. Since the robustness to certain variations is not generalizable across models, the thesis concludes that a continuous evaluation of new systems is necessary to understand their particular strengths and weaknesses. Only by a comprehensive evaluation can we make a fine-grained decision of which LLM aligns best with the given requirements of a *C-to-Rust* migration.

Acknowledgments

Firstly, I would like to express my sincere gratitude to Robert Bosch GmbH, and specifically to all people involved for choosing me for this interesting thesis topic. Without me being given this opportunity, this thesis would not have happened. Likewise, I am very grateful to Prof. Andreas Schmietendorf, who immediately made himself available to supervise this topic from the university's side. Moreover, I want to express my greatest gratitude for his remarkable responsiveness throughout the time of the thesis. This enabled a consistent and reliable supervision even through geographical distance. His valuable feedback on the general structure significantly benefited this thesis, and I am also very thankful for the collegial manner in which the conversations were held.

Furthermore I am incredibly thankful to have had Jesko Hecking-Harbusch and Matthias Woehrle as my supervisors at Bosch. Working with them was an incredibly educational and joyful experience. The thesis benefited tremendously from both their experiences and honesty, which I was able to utilize from countless discussions and meetings. From the beginning to the end, I felt truly supported, and each question, either small or complex, was answered with high detail and always instantaneously. The time at Bosch was truly special and I am very glad for this incredibly beneficial supervision, which not only was great on a professional level but also on a human level which led to a very comfortable working environment. I am also very thankful for all the actions that led to an invention report, as well as the incorporation in a research paper that relied on the thesis's experimental data. Although it has only been six months, the time working in this group has been highly informative and will likely sustainably impact me going forward. Additionally, a special thanks goes to Jesko Hecking-Harbusch again for iteratively reviewing this thesis, leaving feedback and opinions which shaped the result to what it has finally become. I am very grateful for getting this extra amount of attention, which is clearly not self-evident. Moreover, I want to thank all other colleagues at Bosch from whom I directly or indirectly benefited, which not only contributed to making the time very pleasing, but also gave answers to technical questions or set up the experimental environment (Jochen Quante, Martin Leinberger, and others).

Beyond the professional support, I am also incredibly thankful for my entire family, who supported me throughout the time of this thesis and, of course, throughout my entire life. Without their generous and ubiquitous support, neither this thesis nor my degree would have been possible. I am truly blessed to have such great people around me, on whom I can always rely. Specifically, regarding the thesis, their support during my move to and my life in the Großraum of Stuttgart was invaluable, as the entire family did not hesitate to assist me in every possible way.

Lastly, I am very grateful for my loving and supportive girlfriend, who encouraged me to write this thesis here in Renningen, even though it meant managing a long-distance relationship with many video calls.

Contents

List of Figures

List of Tables

Abbreviations

APPS	*Automated Programming Progress Standard*
API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
LLM	*Large Language Model*
AI	*Artificial Intelligence*
SOTA	*State of the Art*
FFI	*Foreign Function Interface*
JSON	*JavaScript Object Notation*
IEEE	*Institute of Electrical and Electronics Engineers*
SE	*Software Engineering*
Seq2Seq	*Sequence to Sequence*
MBPP	*Mostly Basic Python Problems*
NLP	*Natural Language Processing*
NN	*Neural Network*
RP$_s$@k	*Robust Pass$_s$@k*
RC$_s$@k	*Robust Change$_s$@k*
RD$_s$@k	*Robust Drop$_s$@k*
RR$_s$@k	*Robust Relative$_s$@k*
RQ	*Research Question*
UMAP	*Uniform Manifold Approximation and Projection*

1 Introduction

1.1 Motivation

"If the translator does his job as he should, he is a benefactor of humanity; otherwise he is a veritable public enemy."
Miguel Sáenz, as cited in [MW21]

Although Sáenz was talking about human language, his words are highly relevant for today's *Artificial Intelligence* (AI)-based systems that can perform automated code translation. An incorrect or non-robust automatic translator can quickly become Sáenz's *"public enemy"*. Think of an automatic system that translates a safety-relevant software function correctly. However, as soon as a developer adds a supposedly harmless comment or performs a formatting change, the automatic system unexpectedly starts generating faulty or unsafe code. Such inconsistent translations might be undetected, especially when the system normally has a high success rate. This could ultimately lead to serious errors in safety-relevant systems and reflects Sáenz's statement that a translator has to do *"his job as he should"*, and consequently demands the translator to do their job robustly.

The rapid advancements in *Large Language Models* (LLMs), primarily caused by the introduction of the *transformer* architecture [Vas+17], led to the highly popular LLM's we know today: *ChatGPT* [Ope22a], *Gemini* [Ani+23], or *Llama* [Tou+23]. LLMs and their *emergent abilities* in processing natural language [Wei+22] sparked a large interest among the scientific community, and is highly supported by the economy, and various AI-based startups. Altogether, they revealed a vast amount of interesting use cases for generative AI [Zha+23c].

One of these is generative AI for *Software Engineering* (SE), which has the potential to revolutionize its area [Hou+24; Fan+23]. Fan et al. name scenarios like code generation, software testing, document generation, and many more, that are active research areas in this domain [Fan+23]. Furthermore, the acceptance among the developer community for AI in SE apparently is very high. A survey found that 92% of US-based developers already use AI coding tools in and outside of their work [Sta23].

However, this enthusiasm should be approached with caution. Many publications indicate that LLM-based coding tools can introduce low-quality code [Par+24], containing bugs [Per+23; Jes+23] or security vulnerabilities [Sch+21; ANA23].

Furthermore, a recent paper points out that good performance on *State-of-the-Art* (SOTA) benchmarks does not necessarily mean that a model is "intelligent" enough to generalize well to other similar tasks in real-world environments [Nez+24]. Specifically, Nezhurina et al. show that even the newest and largest SOTA models have problems with simple tasks that require logical thinking and common sense. They conclude that it is necessary to revisit current benchmarks and call for a re-evaluation of the models' capabilities.

A possible interpretation of their finding is that LLMs have to be evaluated specifically for their tasked problem domains, since results from other domains may not directly transfer to seemingly similar problems, which thus could lead to a general overestimation of an AI system's capabilities.

A promising problem domain is automated code translation [Eni+24; Yan+24b; Zha+24a], especially the translation from *C* to *Rust* [SS24; HR25; DAR24]. *C's* remaining dominance in systems-level and safety-critical software [BBH18; San20] and *Rust's* advantage in memory safety [Jun+18] pose a great potential for migrating *C* codebases into *Rust*. Since *Rust* eliminates some of the most critical classes of runtime errors common in *C*, a migration from *C* to *Rust* is highly desirable. This gets underlined in an article of 2019, where *Microsoft* reported that 70% of their security issues stemmed from memory-safety problems [Cim19]. Many of those could be prevented only by using the safety features that come with *Rust*. However, manually rewriting and translating *C* codebases is impractical.

While there are tools like *c2rust* [Imm19a] that provide a rule-based, syntactic translation, the resulting code often remains unsafe, as it structurally mirrors the original *C* code. Specifically, the handwritten, rule-based translators fail to leverage *Rust's* idiomatic safety features [Yan+24b; Emr+21; Li+25]. By contrast, LLMs trained on large amounts of data, including idiomatic and safe *Rust* code, offer an innovative method to produce more idiomatic translations [Yan+24a]. However, this opportunity comes with introducing the risks and weaknesses posed by LLM-generated code.

Fortunately, the nature of code translation offers an advantage that other AI for SE use cases do not have. The advantage is that the output can be directly verified for correctness. An LLM-based *C* to *Rust* translation system must ensure that the generated *Rust* code (*i*) compiles without errors and (*ii*) is functionally equivalent to the original *C* code.

Both of which can be verified in a generate-and-check pattern [Als+24]. Whether the translated file compiles can be checked by the *Rust* compiler, whereas the functional equivalence can be verified by *differential fuzzing* [Eni+24]. In addition, the automatic verification can be utilized for another benefit. Specifically, the checking results can be leveraged to implement a *feedback loop* process that directly causes a re-prompting of the LLM, in case of failure. This re-prompting gives feedback to the model on why a proposed solution failed. Such a feedback-based approach mimics human interaction and might improve the likelihood of a successful translation.

While this generate-and-check pattern presents a powerful way for verifying functional equivalence in a specific translation attempt, it does not necessarily prevent the translator from becoming Sáenz's *"veritable public enemy"*. The generate-and-check pattern might enable the assessment of the system for specific inputs, but it does not show how reliably the system creates correct translations under realistic conditions. Real-world codebases are diverse. They are a product of multiple authors, with varying styles, commenting levels, or logical expressions (e.g., `for` vs `while` loops). An LLM-based translator that aims to be a *"benefactor"*, rather than a potential *"enemy"*, has to be robust against such diversity. It needs to produce compilable and functionally equivalent Rust code, not just for one specific input but also when that input mirrors these common, semantically similar variations.

However, the demand for robustness must consider that LLMs are stochastic models, whose outputs are influenced by sampling strategies during generation [Hol+20]. Therefore, it is unreasonable to expect completely deterministic behavior with always identical outputs, since that is not how these models work. Thus, some extent of fluctuations must be accepted in the system. What this means in detail is that there can be repeated runs with the same or semantically similar inputs that lead to success as well as failure of the system. Although not the ideal scenario, it is considered acceptable as long as it remains in a statistically expected range reflecting the model's probabilistic nature.

The actual problem, where LLM-based translators risk becoming Sáenz's *"veritable public enemy"*, is a significant lack of robustness. More precisely, this means an unexpected change away from the usual performance of a translation system when it is prompted with semantically similar, but slightly altered inputs, which also represents the diversity of the real world. Such deviations suggest a potential weakness in the robustness of the model to certain inputs.

As a result, to deploy or sell such systems with confidence, they must be systematically assessed for *robustness*. It needs an evaluation of whether the LLM-based translator performs consistently under real-world input variations, and therefore acts as *"benefactor"*. Solely testing the code translation capabilities concerning success rate is not enough to assess the systems confidently for practical applications.

An often adopted method to measure and quantify such specific characteristics is by creating standardized benchmarks and evaluation frameworks. The rise of public and standardized benchmarks has had a tremendous impact on the recent progress in *machine learning*. The open comparison with benchmarks like *ImageNet* [Rus+15] or *GLUE* [Wan+19] led to rapid progress in *image processing* and *natural language processing*. A comprehensive robustness evaluation framework for LLM-based code translation could create a similar leap, as this reveals *robustness* as another important aspect for LLM evaluation, beyond standard performance metrics. Only by revealing weaknesses can they be proactively improved.

Therefore, a meaningful benchmarking framework must not only evaluate the general performance of the system, but also its reliability and robustness in such scenarios. In addition, it is a key challenge to differentiate between the inherent nondeterminism producing expectable fluctuations, and true robustness deficits that cause drastic changes to the reliability of the performance. Only demonstrating that an LLM can translate *C* to *Rust* for exemplary data is insufficient and could potentially overestimate the model's capabilities, producing the same misconception highlighted by Nezhurina et al. [Nez+24].

This motivates the development of a comprehensive evaluation framework that incorporates these ideas. Such a framework should (*i*) apply a wide range of realistic and meaningful code variations through perturbations, (*ii*) provide methods to separate noise from robustness deficits, (*iii*) consider the influence of approaches like *feedback loops*, and (*iv*) investigate the relationship between robustness and input similarity. The development and application of such a framework for *C* to *Rust* translation is the focus of this thesis.

1.2 Problem Statement and Research Gap

The motivation highlights the need for assessing LLM-based translators on robustness. While there are publications that examine the robustness of LLMs in SE-tasks [Wan+23; Yan+23a; Mas+23; Imp+25], closer investigation reveals the following significant gaps concerning the motivated evaluation in Section 1.1.

1.2.1 Robustness to Instruction vs Robustness to Code

Current research on robustness evaluation mainly studies variations in the natural language part of LLM-prompts [Yan+23a; Mas+23; Imp+25]. While this is relevant for code generation based on problems or requirements, it is less meaningful for the case of code translation. In this case, the LLM prompt consists of a natural language part and a code part that is to be translated into the target language. The natural language part describes the general task to the model, e.g., “Translate the following C code into Rust.”. A deployed code translation system will not vary in its underlying task description, rather, it will vary in the code that is to be translated. Consequently, a relevant robustness evaluation for code translation has to focus on input variations on the code part of the prompt. Specifically, such variations should reflect input variations that simulate the diverse characteristics of codebases, including different formatting rules, styles, detail, or language of comments, variable names, or a refactored control flow. While Wang et al. [Wan+23] include code variations in their robustness evaluation, the evaluated code variations are primarily on a superficial level, only sparsely testing structural variations. Furthermore, they focus on problem-based code generation and code completion, and not specifically on code translation, which reveals another research gap.

1.2.2 Robustness when Translating Code

As far as can be determined from the literature, no work evaluates robustness specifically for code translation. Recalling the interpretation of Nezhurina et al. [Nez+24], the findings of benchmarks of other domains should not be translated to a different, seemingly similar domain. The prior evaluated robustness for problem-based code generation or code completion has some fundamental differences from code translation.

1. In problem-based code generation, the task is to fulfill or solve a specific problem, defined in natural language, which is not necessarily complete or unambiguous. In code translation, a certain code in the original language has to be translated into the target language, which is unambiguous. Consequently, input variations in code translation maintain its unambiguity, and subsequent robustness deficiencies can be directly attributed to the model’s incapability of handling such variations, rather than to ambiguities in the task itself.

2. Other domains require the LLM to generate Code from scratch, whereas code translation includes the prior logic and control flow that has to be converted into the representation of the target language. Not generating from scratch may reveal other robustness capabilities, as it does not require reasoning about an algorithmic solution, but rather an idiomatic conversion.

3. Code generation is commonly verified with unit tests [Wan+23; Mas+23], whereas the goal of code translation is functional equivalence. A proper functional equivalence verification reveals even the smallest functional differences, which might be unnoticed by unit tests. As a result, a robust code translation system must reliably produce code that passes the strict functional equivalence check.

This incomplete overview highlights that robustness knowledge from the other domains cannot be directly transferred to code translation. Therefore, there is a clear need for a code-translation-tailored robustness evaluation.

1.2.3 Programming Languages and Benchmark Complexity

So far, the literature on the robustness of code generation focuses mostly on popular languages, such as *Python* [Wan+23; Yan+23a] or *Java* [Mas+23]. Consequently, these languages offer more source code an LLM could be trained and evaluated on. LLMs robustness in generating less popular, or low-level, programming languages is underrepresented in the literature. Moreover, it is also noteworthy that some of the used benchmark datasets have been criticized for being too simple and not reflecting real-world programming scenarios [Aga+24; Sid+24; YBS24]. Therefore, a robustness evaluation with actual relevant code and with a focus on low-level languages like *C* and *Rust* remains unexplored.

1.2.4 Model Modernity

Considering the rapid pace of LLM improvements, the related robustness evaluations assessed LLMs that are no longer SOTA. According to standardized benchmarks for SE tasks like problem-based code generation, the capabilities of LLMs have improved significantly in recent years [Zhe+23; LM24]. While the first code-related LLMs struggled with basic programming problems, today's models obtain much higher correctness scores. In some cases, modern LLMs even produced better results than a human reference [HJ24]. That raises the question of whether prior conclusions about robustness remain applicable to the models used today. Although knowledge about increased performance should not be taken as evidence of increased robustness, the true robustness of modern models can only be understood by explicitly testing for it. Hence, an evaluation with modern SOTA LLMs needs to be conducted.

1.2.5 Unexplored Aspects of Robustness

Besides the clear research gaps that mostly arise from previous works not being up-to-date or focusing on other programming languages and use cases, there are nuances of robustness that are undiscovered in related robustness evaluations.

1. The *generate-and-check* pattern for code translation allows *feedback loops* for correcting incorrect translations in iterative cycles. Since the robustness of code translation has not been evaluated yet, the impact of *feedback loops* on robustness is also unexplored. Do *feedback loops* resolve robustness issues, or do they only affect the general performance of the translation system?

2. Current robustness works do not contain a methodology to distinguish the model's inherent noise and genuine deficiencies in robustness. Wang et al. [Wan+23] bypassed this problem, by utilizing *greedy-sampling*, that reduces the nondeterminism of LLMs and will be explained in Section 2.2.4. However, this could hinder the models from showing their real potential and is therefore not commonly used in practice [Hol+20]. A robustness evaluation relevant for practical application has to account for the "normal" stochastic nature of LLMs.

3. Current publications investigated the similarity of the input variations that are used in the robustness evaluation, for the sake of proving their semantic similarity and naturalness [Wan+23; Mas+23; Imp+25; Yan+23a]. However, they did not investigate whether input similarity correlates with robustness. The intuition would be that less similar input variations cause stronger robustness deficits than highly similar inputs. If this is a real phenomenon remains to be explored.

This overview details that there is a significant research gap for the systematic and comprehensive robustness evaluation of modern LLMs in the context of *C-to-Rust* code translation, which calls for an evaluation framework that addresses all the mentioned aspects and provides a differential analysis of the different nuances of robustness.

1.3 Research Questions and Contributions

The previous sections gave the information to detail the necessities required to explore the existing research gaps. Specifically, the goal of the thesis is the development and application of a comprehensive robustness evaluation framework that enables a multidimensional analysis of the relevant aspects when translating *C* code into *Rust*. These dimensions reflect *(i)* the sensitivity of the code translation system to various input variations, *(ii)* differentiation between nondeterministic fluctuations and genuine robustness deviations, *(iii)* the *feedback loops'* impact on robustness, and *(iv)* the correlation between input similarity and performance deviation.

This raises the discussion of the following main RQ:

RQ1: "What methodologies and components should be integrated into a comprehensive evaluation framework to assess the robustness of an LLM-based code translation system?"

The upcoming chapters of the thesis elaborate a systematic methodology that presents components that are part of such an evaluation framework. This includes the identification and implementation of a diverse set of perturbation strategies reflecting real-world relevant input variations. This involves strategies that have been established in robustness evaluation works or strategies that have been introduced in another domain, such as *source code plagiarism detection* or *data augmentation*. Moreover, the thesis proposes additional input variation techniques. To quantify the robustness the framework applies a robustness metric of Wang et al. [Wan+23] and introduces a new metric. By applying the proposed framework on an existing SOTA code translation system [QHW25] with well-established LLM (i.e., *GPT-4o-mini* [Ope24a]) and testing it with a dataset including real-world code snippets, the thesis examines whether the presented methods enable a comprehensive robustness evaluation.

The implementation and application of such a framework is the main objective of this thesis. However, the components of the framework can be leveraged to examine in-depth aspects of robustness that have received little attention so far. These aspects are addressed by the following *Research Questions* (RQs):

RQ2: "How does one differentiate between inherent LLM nondeterminism (noise) and true robustness deficits?"

The inherent stochasticity of LLMs can lead to varying performance observations. A model might succeed and fail for the exact same input in repeated attempts. These nondeterministic fluctuations, or "noise", must be accounted for in a robustness evaluation.

A simple comparison of unperturbed results with perturbed results risks misinterpreting normal performance fluctuations as non-robust behavior. To understand what performance deviations are normal for a model and what deviations are genuinely caused by varying inputs, a comprehensive robustness evaluation has to address this question.

Without making this distinction, the evaluation cannot assess whether drops in performance are a systematic weakness related to specific input variations or simple statistical artifacts.

The methodology of this thesis introduces a statistical approach that tackles this specific question and enables a more detailed analysis of an LLM's robustness to input variations.

RQ3: "Does incorporating a *feedback loop* strategy impact robustness?"

The generate-and-check characteristic enabling *feedback loops* in code translation raises the question whether these *feedback loops* impact the robustness of such systems. Therefore, the thesis examines whether and how the approach influences the robustness of a code translation system, contributing knowledge that has not been explored before.

RQ4: "What is the correlation between semantic similarity and perturbation-based robustness?"

By correlating the translation success with input similarity between the perturbed and unperturbed input, the thesis investigates whether the robustness to specific perturbations can be predicted through similarity checks or whether the models remain entirely unpredictable. Recall that the intuition is that more similar inputs should yield more robust results than perturbations with stronger variations. Addressing this RQ verifies whether this intuition holds.

RQ5: "Are robustness results consistent using different LLMs?"

Lastly, the thesis investigates whether the conclusions about specific aspects of *GPT-4o-mini's* robustness transfer to other SOTA LLMs (*GPT-3.5-turbo* [Ope23], *Phi-4* [Abd+24], and *Qwen2.5-Coder-14B* [Hui+24]) and what implications this has for LLM-based code translation.

1.4 Outline

Chapter 2 presents the theoretical foundations that are relevant for the understanding of the thesis's methodology and experimental design. This involves fundamentals about traditional code translation, as well as the basic functionality of LLMs. Furthermore, the chapter explains common evaluation benchmarks, metrics, and findings. In addition, the chapter details SOTA LLM-based code translation approaches and verification techniques. Moreover, it provides a formal definition for robustness that is the foundation for the robustness evaluation in this thesis.

Chapter 3 gives an overview of the current state of SE-related robustness works, which are the basis for the proposed methodology of this thesis. This also includes a critical discussion of existing research that underpins the identified research gaps.

Chapter 4 explains the functionality of the proposed framework. Moreover, it details how to use the framework to examine each RQs.

Chapter 5 displays the design of the experiments, including the dataset that is to be translated. Furthermore, it investigates *GPT-4o-mini's* baseline performance, which is later used as a reference to assess robustness.

Chapter 6 examines the general robustness of *GPT-4o-mini*, without focusing on the nuanced aspects of robustness. The chapter elaborates, whether the LLM exhibits non-robust behavior when the system is tasked with the incorporation of *feedback loops*.

Chapter 7 analyzes if the *feedback loops* affected the robustness conclusions in Chapter 6 and therefore empirically investigates RQ3.

Chapter 8 looks at how input similarity relates to the success of a translation with *GPT-4o-mini*. It therefore extends prior results of Chapter 6 and Chapter 7 to gather information for RQ4.

Chapter 9 reveals whether the findings of earlier chapters apply to the other LLMs, enabling RQ5 to be discussed. The chapter concludes whether robustness should be another dimension to the already multi-factored choice of the right LLM, including *cost*, *accessibility*, and *performance*, or if robustness results are similar across models.

Chapter 10 summarizes the observations of chapters five to nine and presents the answers to the RQs. Additionally, the chapter reflects on threats to the validity of the results and also gives an outlook for future research. By summarizing all information, the chapter can give a clear answer whether modern LLMs are a *"benefactor"* or rather a *"veritable public enemy"* when they are tasked to translate *C* code into *Rust*.

2 Foundations for LLM-based Code Translation

This chapter presents the theoretical foundations necessary to follow the rest of the thesis. This includes information about characteristics of *C* and *Rust* in Section 2.1, that justify the need for a *C-to-Rust* translation. Moreover, this section presents the functionality and drawbacks of traditional code translation systems. This leads to a brief introduction of the most important concepts of LLMs, as well as use cases and pitfalls of AI-assisted *software engineering* in Section 2.2. To only present the relevant concepts of LLMs together with their original publication, the chapter relies on extensive LLM surveys that introduce LLM functionalities and reference initial publications [Zha+23c; Zhe+23; Fan+23]. To build a foundation for the evaluation of LLM coding capabilities, Section 2.3 highlights common evaluation benchmarks and explains the most relevant SOTA performance metric. Lastly, Section 2.4 details approaches for LLM-based code translation, which also includes presenting relevant LLM-based code translation systems that leverage the *generate-and-check* pattern mentioned in Chapter 1. Therefore, the section also includes a thorough explanation of how the translation can be verified and how this verification can be leveraged for an automatic refinement process with *feedback loops*.

2.1 Fundamentals of C-to-Rust Code Translation

Institute of Electrical and Electronics Engineers (IEEE) [IEE90] defines SE as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software". Thus, it includes the consistent improvements and modernizations of legacy software to reduce their susceptibility to errors.

2.1.1 Why C to Rust

While the focus of this thesis is the robustness of LLM-based code translation, it is necessary to reason why translating from *C* to *Rust* is highly desirable, and why traditional automatic approaches fail when doing this.

According to a research project of *DARPA*[1] "the software engineering community has reached a consensus" to prefer memory-safe languages, "that can reject unsafe programs at compile time" [DAR24].

Memory Safety Challenges in C

The problem with *C* is its susceptibility to memory safety issues [DAR24]. That means if a program accesses memory that it is not supposed to, the program reacts in an undefined way [AHP18]. This could potentially lead to crashes, but can also show in corrupted data

[1]The Defense Advanced Research Projects Agency (*DARPA*) belongs to the United States Department of Defense and supports relevant research projects, see: https://www.darpa.mil/

or exploitable vulnerabilities. There are multiple sources highlighting that many of today's security vulnerabilities go back to these memory issues. For example 60-70% in *iOS* and *macOS* environments [Keh19], 70% in the products of *Microsoft* [Cim19], and *Google* estimates that 90% of *Android* vulnerabilities are a result of memory safety issues [Zha19].

While memory safety can be achieved through runtime checks in *C*, it comes with the loss of performance and is therefore mostly not deployed in production [Li+25; Nag+09; Nag+10].

Memory-safe languages work inherently differently and can completely prevent this class of errors [DAR24]. Instead of causing unpredictable behavior, these languages are designed to catch illegal memory access at runtime [Jun+18; Gro21] (e.g., *Java's* `ArrayIndexOutOfBoundsException` [Oraa] or more precisely *Rust's* runtime panic on out-of-bounds access [The21].). That means memory-safe languages "can neither affect nor be affected by unreachable parts" [AHP18] of memory.

According to Jung et al. [Jun+18], system-level languages like *C* often provide low-level resource management at the expense of guarantees of memory safety and therefore involve a trade-off. However, *Rust* addresses this trade-off by employing an *ownership* and *borrowing-based* type system that is utilized for `safe` code, but can also be bypassed in case of precise low-level control [Jun+18; MI14].

Rust's Memory Safety Mechanisms

Rust solves these issues primarily through its *ownership* and *borrowing-system*. In *safe Rust*, the compiler ensures at compile time that every value has a unique owner and that all accesses strictly align to predefined lifetime rules [MI14]. This prevents pointers from referring to deallocated memory or concurrent write access, which can cause unpredictable side effects [MI14]. Furthermore, *safe Rust* incorporates runtime checks. These checks prevent undefined behavior for errors that are not covered by the *ownership* system, such as *index out-of-bounds* access [Zha+22]. Instead of undefined behavior, this results in controlled panics. These are less harmful because they provide a defined error state that may terminate the program, subsequently preventing it from continuing with corrupted memory.

However, *unsafe Rust* allows developers to bypass these compile-time and runtime safety guarantees to perform low-level operations, but again with the cost of the programmer actively preventing issues [Jun+18]. While it is possible to write memory-safe *unsafe Rust*, like in *Rust's* standard libraries [Jun+18], translating the *C* code into *safe Rust* is the preferred and idiomatic goal whenever feasible.

2.1.2 Code Translation Task

The *code translation* task, also described as *transpilation*, is a specific form of software maintenance, where an existing codebase is converted into another, newer programming language, to improve attributes like safety, maintainability, or performance, while maintaining the same functionality, i.e., *functional equivalence*.

The Requirement of Functional Equivalence

The key requirement and ultimate goal of any code translation is maintaining the *functional equivalence*. In detail, this can be defined as "the property of two functions to produce the same outputs when given the same inputs, yielding the same observable behavior, even if their implementations differ syntactically." [MVC24]. A code translation is only correct when this characteristic is met, which stresses why the validation of this behavior is a crucial part of validating code translation.

Traditional Transpilation Techniques

Before the age of LLMs, automated code translation primarily relied on rule-based transpilers [Gui+24; Yan+24b]. These act similarly to compilers, but instead of generating machine code, they generate source code [IN16].

According to Tomassetti [Tom20], transpilers act in three stages.

1. **Parsing stage**: The source code in the original language is analyzed by a parser and transformed to an *Abstract Syntax Tree* (AST) [Aho+14] representation.
2. **Transformation stage**: This is the core logic where the actual translation is performed. It involves traversing the source AST and applying predefined transformation rules to convert the source AST into an equivalent AST of the target language. In difficult translations, there can be more than one iteration of this step.
3. **Generation stage**: The translated AST representation of the target language is generated into the source code of the target language.

These transpilers present a way of translating languages of similar abstraction levels [IN16], but they struggle when abstraction levels change, and code is not directly transferable, which results in unidiomatic code [Yan+24b; Liu+25].

Specific Challenges in C-to-Rust Translation

While *C* and *Rust* share roots in imperative systems, they are not trivially translatable. Legacy *C* code uses patterns like manual memory management, that have no direct safe equivalent in *Rust's ownership* model [Emr+21]. The traditional approaches are not designed to analyze and understand the properties of the *C* code and produce an equivalent *Rust* implementation that adheres to the *ownership* and *borrowing system*. Instead, tools like *c2rust* [Imm19b] simply mirror the existing *C* into an *unsafe Rust* version that bypasses the beneficial safety features [Emr+21].

Emre et al. [Emr+21] found that only 11% of raw *C* pointers can be converted to *safe Rust* by rule-based transpilers, which highlights the difficulty of transferring *unsafe C* into *safe Rust* [Li+25]. In addition, Li et al. [Li+25] name other *C* patterns that are forbidden in *safe Rust*. Specifically, *(i) mutable global variables* and *(ii) unions* [GNUb]. Other publications focusing solely on translation capabilities also name *pointer aliasing*, *switch-case statements*, *goto statements*, or *macros* that are not trivially translatable [PG24; Cai+25].

Consequently, not every *C* file is directly mirrorable to *Rust*, without a real understanding of the logic and semantics of those files.

In comparison, a study examined the translation capabilities of non-expert undergraduate *Rust* users [Li+25]. They showed that most of the participants succeeded in creating reasonable, *safe* translations. However, almost all translations lack a complete functional equivalence. This might suggest that it is easy to convert most *C* functionality, but producing a 100% functional equivalent translation is tedious.

This motivates the exploration of alternative solutions, like LLMs, with the desire to automatically generate translations that are idiomatic and functionally equivalent, without the tediousness of manual translation.

2.2 Large Language Models

LLMs with their ability to process and generate human-like texts present a promising alternative to the traditional transpilers. Instead of relying on handcrafted translation rules, LLMs operate based on statistical patterns they learned during the training on large datasets [Zha+23c]. Specifically, the language models produce the most likely continuation as output, based on the patterns they learned in training on large datasets. These models can have hundreds of billions of trainable parameters and show "emergent abilities" that go beyond simple pattern matching [Wei+22], which explains the broad interest. Since these models are the fundamental focus of this thesis, the following section explains the most important concepts to understand how these models work, which inherent tools can be leveraged for the robustness evaluation and what pitfalls have to be considered.

2.2.1 From Text to Numeric Representations

Since LLMs are statistical models, they can not directly work with text and require numerical representations to perform mathematical operations [Zha+23c]. Therefore, the given input text sequence has to be encoded into such a numerical representation. This conversion heavily relies on these key concepts: *(i) tokenization*, *(ii) embeddings*, and *(iii) positional information*.

Tokenization

This step aims to convert the raw input text into small sequences of characters, called *tokens* [Zha+23c]. While the intuition might be to simply encode single words into numerical representations, the evolution of language models has shown that the tokenization approach is superior. In detail, encoding single words would require defining a specific vocabulary that maps words to unique numbers. However, this vocabulary cannot be infinitely large and therefore will result in an *out-of-vocabulary* problem, for words that are not in the vocabulary. Similarly, it introduces the challenge, how to handle rare words, as the frequency of words would impact the statistical modeling [SHB16].

Modern LLMs therefore use sub-word tokenization algorithms like *byte-pair encoding*, which iteratively builds a vocabulary by repeatedly merging the most frequent pair of adjacent symbols found in the training dataset[2] into a new token [SHB16]. This separation allows encoding every word, as it is either a result of sub-words or can be produced by single characters. Furthermore, the frequency aspect of the encoding reduces the raw word problem, as rare or unknown words get represented by sequences of more frequent sub-word tokens.

Figure 2.1 displays the tokenization for an exemplary sentence. It details that "LLM" itself is not represented as a single token, whereas "->" is.

[2]This training dataset is not necessarily identical to the dataset used to train the model's parameters, but it needs to be large to ensure a representative encoding of the most frequent pairs.

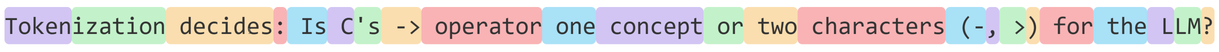

Tokenization decides: Is C's -> operator one concept or two characters (-, >) for the LLM?

Figure 2.1: Visualizing an exemplary tokenization using *OpenAI's tiktoken* [Ope25c] with their *o200k_base* vocabulary.

Embeddings

With tokenization, all words can be encoded into a numerical representation. However, this gives no information about their semantics. For this, an *embedding* model comes into play. Early approaches utilized learnable lookup tables [Ben+03] and later publications used dedicated *Neural Networks* (NNs) to learn these representations more effectively [Mik+13a]. These models encode the token values into high-dimensional vector representations that mirror their semantics. Consequently, similar tokens show similar vectors [MYZ13].

This similarity can be quantified using *cosine similarity* [SWY75a]. This measures the cosine angle between two similar dimensional vectors v_1 and $v2$ [LH13]. Therefore, it uses the dot-product of two normalized vectors, i.e., Equation (2.1).

$$cosine\_similarity(v_1, v_2) = \frac{v_1 \cdot v_2}{||v_1|| * ||v_2||} \tag{2.1}$$

A *cosine similarity* close to one indicates a high similarity, as the vectors point almost in the same direction, whereas minus one indicates vectors pointing in opposite directions. This geographical representation allows mathematical operations that reflect semantic relationships. A well-known example that shows the intuition behind embedding vectors and similarity is:

$$v_{king} - v_{man} + v_{woman} \approx v_{queen}$$

where v is the encoded vector representation of the word, and the resulting vector of $v_{king} - v_{man} + v_{woman}$ has a high *cosine similarity* to the vector of v_{queen}. [MYZ13; Mik+13b].

This concept of semantic similarity is a crucial part of RQ4: "What is the correlation between semantic similarity and perturbation-based robustness?" and is therefore important to keep in mind for the robustness evaluation framework.

Positional Information

By encoding text into tokens and mapping it to representational embedding vectors a key information gets lost, namely the original order of tokens. However, this is crucial for the semantics of the text (e.g., compare "C to Rust" vs "Rust to C"). Therefore, the token representation is extended by positional information [Zha+23c]. Various methods have been proposed for this [Zha+23c], and recent models commonly rely on *Rotary Position Embeddings (RoPE)* [Su+24]. A technique that encodes the position by applying position-based rotations to the embedding vectors.

These numerical representations, reflecting semantics and positional information of tokens, form the input for the underlying neural network architecture, which processes this information.

2.2.2 Transformer Architecture

Modern LLMs are predominantly based on the *transformer* architecture proposed by Vaswani et al. [Vas+17]. This architecture significantly outperformed previous *Sequence* *to Sequence* (Seq2Seq) approaches based on *recurrent neural networks* [Elm90] and *long short-term memory (LSTM)* [HS97].

Prior Seq2Seq approaches had two major problems *(i)* fixed-sized input vectors that prevented the models from processing long input sequences and led to the models forgetting information from early steps. *(ii)* the sequential approach could not easily be parallelized, as each step relied on the states of previous steps, which resulted in inefficient model training [Vas+17].

The *transformer* solves these problems by its *self-attention* mechanism [Vas+17]. This mechanism enables the model to weigh the importance of tokens for a specific context, which subsequently reduces the problem of forgetting information and also enables the processing of all tokens of the sequence in parallel.

The original transformer incorporated an *encoder-decoder* approach, where the *encoder* processes the input and refines it into an internal representation, which is then used by the *decoder* to generate the output token by token. This decoder uses a specialized version of attention, *masked self-attention*, which only allows to attend to previous tokens [Vas+17].

Popular generative LLMs of today (e.g., *GPT* series [Bro+20; Rad+19], *Phi-4* [Abd+24], or *Llama* [Tou+23]) primarily use a *decoder-only* architecture [Zha+23c]. For the generation of text, this is preferred because it eliminates the need to train a separate encoder, which reduces computational and memory costs while producing comparable or even better performance.

2.2.3 Learning on a Large Scale

While the *transformer* architecture is the key component of modern LLMs, its promising capabilities are only due to a large-scale learning process on massive datasets. This process involves two steps *pre-training* and *fine-tuning* [AG24].

Pre-Training

The unsupervised *pre-training* process builds the foundation for these LLMs and requires the majority of computation and training time [AG24]. In this step, the model is trained on massive text-based datasets, which enables it to learn concepts of grammar, context, or language patterns [AG24; Zha+23c]. A common objective of this training step is *language modeling*, which refers to assigning probabilities to sequences of words and predicting upcoming words [JM25; Zha+23c].

A publication of Kaplan et al. [Kap+20] empirically showed that the performance of these models heavily depends on scaling, i.e., number of trainable parameters of the model. Larger models, trained on larger datasets, with more computational power, yield better results. In addition, at a certain scale, these models showed “emergent abilities”, abilities that are only present in large models and not in small models [Wei+22]. Both findings are the core motivation for why modern models strive to increase the scale of both model size and training data.

Fine-Tuning

After the large-scale *pre-training*, the model is fine-tuned to a narrower task reflecting the desired use case [AG24]. A noteworthy example is the *fine-tuning* of *GPT-3* [Bro+20] to the initial version of *ChatGPT* [Ouy+22; Ope22b]. Specifically, this step involved *reinforcement learning from human feedback (RLHF)*, which tuned the model to align more to human preferences [Ouy+22]. Instead of *fine-tuning* all parameters, common strategies like *LoRA* [Hu+22], or *QLoRA* [Det+23], only update certain parameters to reduce computational costs.

2.2.4 Probability-Based Output Generation

The previous sections explained the pivotal concepts of encoding text into semantic numerical representations that are processed by a *transformer*-based architecture and trained on a large scale. To generate their output, LLMs employ *decoding strategies* [Zha+23c].

Since the *language modeling* task trained the model to predict the next token based on the previous tokens, the model yields a probability distribution for likely tokens. The simplest *decoding strategy* is *greedy sampling*, also denoted as *greedy search*. This approach always selects the token with the highest likelihood. However, it has been shown that this strategy is not optimal and tends to create repetitive outputs [Hol+20].

Therefore, various decoding strategies have been proposed, a common one being *temperature-based sampling* [AHS85; Hol+20]. This sampling strategy scales the output probabilities with a *temperature* parameter, which impacts the probability distribution, and then randomly chooses one token based on the probabilities. In detail, low temperature values produce more deterministic results than high temperature values. A commonly used temperature value in the literature is 0.7 [Ouy+22; Hol+20; Rad+19; FLD18].

This randomness factor has to be considered for the robustness evaluation, as it adds a non-deterministic character, which impacts the reliability and consistency of a model's outputs.

2.2.5 LLMs in Software Engineering

The presented concepts form the basis of LLMs. However, this basis was specifically developed for human language and not necessarily for SE and programming. This section details certain aspects that are important for the SE capabilities of LLMs.

Naturalness of Code

As mentioned before, with large-scale *pre-training* and *self-attention*, the models are able to capture concepts like grammar and patterns of natural language. However, similar to the grammar and patterns in natural language, programming languages also involve repetitive patterns and structures [Hin+16]. Hindle et al. [Hin+16] show that software code is rather simple and repetitive, which is highly beneficial for a probability-based estimation of a next-token prediction (i.e., *language modeling*). Consequently, by training LLMs on existing codebases, the models can also learn a next-token probability for coding tasks, by learning the concepts of syntax and semantics of programming languages [Fan+23].

Software Engineering Use Cases of LLMs

Fan et al. [Fan+23] state that LLMs have the potential to impact the entire software development life cycle. One of the most prevalent use cases is plain code generation or code completion, which aims to generate code based on an instruction or comments. A well-known tool for this application is *GitHub Copilot* [Che+21]. Peng et al. [Pen+23] investigated the performance increase of developers when using *GitHub Copilot*. They showed that developers were 56% faster developing a non-trivial programming task than a control group without LLM assistance. Furthermore, a recent survey demonstrates that AI programming assistance has a high acceptance rate among developers [Sta23]. Specifically, this survey states that 92% of US-based developers already use AI coding tools in and outside of their work.

Besides plain code-generation, Fan et al. name publications in *software-testing* [Lem+23; Sch+23], where LLMs are leveraged to generate test cases and improve test coverage. Furthermore, LLMs show promising results in *software-repair* [XZ22; Jia+23], automatically debugging and fixing code based on instructions or static analysis. Moreover, there are works in *requirements-engineering* [Zha+23a; Luo+22] that examine LLMs capabilities extracting requirements out of documents, or detecting inconsistencies in defined documentation.

Key Challenges for LLMs in Software Engineering

While the broad usability and the *emergent abilities* imply a high potential of LLMs, they also have boundaries and drawbacks. A known problem of LLMs in general is *hallucination*, which describes the model outputting false information [Yan+24d; Fan+23]. In the case of SE, the same issues can be found [Ma+23; Fan+23]. Specifically, this means, while the generated code seems plausible, it may be incorrect or contain bugs [Fan+23].

There are several works that show LLMs can produce low-quality code [Par+24; GG24]. Therefore, the creation may be more efficient at first, but with the cost of worse maintainability over time [Zie+24]. In addition, there are works that point out LLMs might produce code with more *code smells* [Liu+24] and higher *cyclomatic complexity* [YOT22].

Besides low-quality code, it has been shown that LLMs can introduce bugs [Per+23; Jes+23]. This may show in syntax errors and code that is not compilable [Tam+25], and therefore is directly noticeable. Unfortunately, LLMs can also produce incorrect logic [Din+23], miss certain edge-cases [Tam+25], or hallucinate methods that call non-existing *Application Programming Interface*s (APIs) [Tam+25].

Furthermore publications have shown that LLM generated code can involve security vulnerabilities [Sch+21; ANA23; Pea+25; Per+23].

These findings highlight that the promising capabilities of LLMs should be enjoyed with caution, which underpins the need for thorough benchmarking and testing of these models.

2.3 Evaluating LLMs in Software Engineering

Publicly evaluating and comparing approaches is a key driver for the tremendous progress of machine learning. Specifically, benchmarks like *ImageNet* [Rus+15], or *MS COCO* [Lin+14] enabled a standardized evaluation for *image processing* and *object-detection* tasks, which, according to public leaderboards, led to approaches with significantly increased accuracy since their release [Cod25a; Cod25c]. The same observation can be made for the *GLUE* benchmark [Wan+19] in *Natural Language Processing* (NLP). Public leaderboards reveal thousands of publications referencing *GLUE*, improving and comparing methodologies [Cod25d].

It is therefore reasonable to expect that public and standardized benchmarks for AI-based SE will accelerate innovation as well. Indeed, already today's leaderboards indicate drastic improvements made in the last few years [Cod25e].

2.3.1 Common Code Generation Benchmarks

There are various benchmarks evaluating AI-based SE tasks. Some of them are also relevant for publications specifically assessing robustness of LLMs, which are explained in Chapter 3.

HumanEval

HumanEval has been introduced by *OpenAI* together with *Codex*, an LLM specifically *fine-tuned* on code [Che+21]. According to Fan et al. [Fan+23], this publication led to an explosion in LLM-based code generation.

In detail, this benchmark consists of 164 python programming problems with unit tests. Each of this problems contains a function signature and a docstring that describes the functionality of the function that has to be generated. The LLMs have to complete the function based on the signature and its docstring explanation.

While *HumanEval* presents a common standard for the evaluation of code generation capabilities, recent publications express criticism. These works mention that this benchmark does not accurately reflect coding capabilities in real-world environments and does primarily consist of coding interview-style problems [Aga+24; Sid+24; YBS24]. Moreover, there are concerns that modern LLMs might have used *HumanEval's* dataset for *pre-training*, which leads to *data leakage* and would prevent a genuine evaluation [Mat+24; Sid+24].

MBPP (Mostly Basic Python Problems)

Mostly Basic Python Problems (MBPP) [Aus+21] is another popular benchmark and is similar to *HumanEval*. It contains roughly 1000 short *Python* tasks, consisting of docstrings and unit tests. In addition, the criticism of *HumanEval* also applies to MBPP [YBS24; Sid+24; Mat+24].

APPS (Automated Programming Progress Standard)

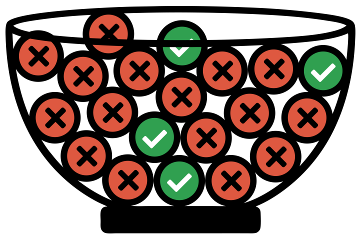
Automated Programming Progress Standard (APPS) [Hen+21] is another common benchmark. It consists of 10,000 *Python* coding problems and unit tests, collected from publicly available coding websites, such as *Codeforces* [Cod], or *Kattis* [Kat]. According to Roziére et al. [Roz+23], the prompts of this benchmark are less direct and more complex in comparison to *HumanEval* or MBPP. However, while APPS generally is more challenging, it is also affected by *data leakage* [Zho+25b] and therefore does not accurately reflect code generation performance for all LLMs.

Others

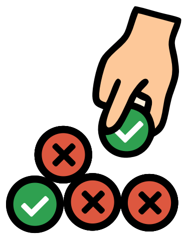
As the focus of this thesis is on code translation, it is worth mentioning that there are also benchmarks specifically focusing on this use case [Yan+23b; Zhu+22]. However, since they are not relevant for current robustness evaluations, they are not discussed in more detail.

2.3.2 Performance Metrics

While there are different performance metrics for different benchmarks, there is one that is highly relevant among benchmarks: *pass@k*. This metric was introduced in *HumanEval* [Che+21] and has since then been used as a standard performance measure and



(a) All $n = 20$ total solutions containing $c = 4$ correct solutions.



(b) Randomly selecting $k = 5$ solutions from the $n = 20$ solutions.

Figure 2.2: Visualization of *pass@k* for an example with $n = 20$ including $c = 4$ correct solutions, when choosing $k = 5$, resulting in *pass@5* $= 1 - \frac{\binom{20-4}{5}}{\binom{20}{5}} = 0.718$.

referenced in various LLM publications [Roz+23; Li+23; Luo+24] and benchmarks [Aus+21; Hen+21; Du+24].

The objective of *pass@k* is to estimate the likelihood for a *successful* generation when performing multiple runs. *Success* mostly refers to passing the unit tests for a given problem, and therefore measures *functional correctness*. However, the metric is not limited to that and can essentially be applied in any context where a binary success criterion can be defined. For example, Zeng et al. [Zen+24] employ a *pass-syntax@k*, a *pass-compile@k*, and *pass-all@k*, that measure the performance in distinct aspects.

In detail, *pass@k* is defined as Equation (2.2) [Che+21]. Given an unordered set of n solutions containing c correct solutions, *pass@k* estimates the probability of selecting at least one correct solution when choosing k samples from all n runs [Che+21; Zen+24]. Figure 2.2 illustrates an example for *pass@5* with $n = 20$.

$$pass@k = \mathbb{E}_x \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{2.2}$$

This metric is the basis for the metrics used in this thesis and for other robustness evaluation metrics that are introduced in Chapter 3.

2.3.3 General Purpose vs. Code LLMs

Chen et al. [Che+21] not only published *HumanEval*, but also *Codex*, an LLM especially *fine-tuned* for code generation. Besides *Codex* there are various other LLMs that were *fine-tuned* [Cha+23] or even *pre-trained* [Li+23] on code.

Therefore, the literature commonly distinguishes between *general purpose* LLMs and *code* LLMs. The *general purpose* LLMs like *GPT-4o-mini* [Ope24a], or *GPT-3.5-turbo* [Ope23], are trained for versatile use cases, including question-answering and programming [Zhe+23], and because of their large size and their *emergent abilities*, they generalize well to programming tasks. Besides these very large models, a recently published smaller model *Phi-4* [Abd+24] also shows promising programming capabilities. The authors used high-quality and synthetic data to yield great capabilities without a very large model size.

LLMs specifically trained on code and programming are referenced as code LLMs [Zhe+23]. Prominent examples, besides *Codex*, are *Code Llama* [Roz+23], *StarCoder* [Li+23], or the recent *Qwen2.5-Coder* [Hui+24]. *Qwen2.5-Coder* is a SOTA open-source LLM that extended the *pre-training* of its base model *Qwen2.5* [Yan+24c] and is available in different sizes (i.e., 0.5B, 1.5B, 3B, 7B, 14B, 32B).

Having *general purpose* LLMs and code LLMs raises the question of which one performs better for programming tasks. Zhen et al. [Zhe+23] conclude that there is no definitive answer to this question. Code LLMs *fine-tuned* to SE-tasks normally outperform their base models [Zhe+23]. In addition, they state that code LLMs mostly outperform the general purpose LLMs with a similar number of parameters.

However, more recent publications show that the large *general purpose* models like *GPT-4* outperform many SOTA code models on certain benchmarks or tasks [Du+24; Niu+24; Guo+24].

2.4 LLM-based Code Translation

This section aims to detail information specifically on LLM-based code translation and gives an overview of SOTA LLM-based *C-to-Rust* translation systems. Section 2.1.2 highlighted common problems with rule-based *transpilers*. Specifically, those often produce non-idiomatic code and also fail to leverage *Rust's* safety features. Therefore, the learning-based LLMs represent an alternative automatic translation approach, which might benefit from learning patterns of *safe Rust* and might be able to produce more idiomatic *Rust* code as rule-based systems.

While the recent advancements of LLMs in SE and programming are promising, an idle LLM translation has no correctness guarantee. By contrast to *rule-based transpilers*, LLMs are probabilistic generators that predict likely code, but not necessarily correct code, and can produce *hallucinations* that might be unnoticed (see Section 2.2.5). Therefore, a verification of translations and a comprehensive robustness evaluation, that is conducted in this thesis, are highly necessary. The section starts by explaining common methodologies for LLM-based code translation and continues with the necessary verification techniques.

2.4.1 Prompting vs Fine-Tuning

The code translation task with LLMs can be approached with two methodologies: *(i) prompting - Zero/Few-Shot* or *(ii) fine-tuning* and training a model specifically for a certain code translation.

Prompting

The prompting approach leverages LLMs generalization capabilities by directly prompting the task of code translation. Specifically, such a prompt consists of a natural language instruction part and source code. The model translates the code based on its vast learned patterns during *pre-training*. In detail, this approach can be executed either by *(i) Zero-Shot* or *(ii) Few-Shot* prompting.

Zero-Shot code translation prompting describes that the prompt only contains a natural language instruction and the code in the original language that has to be translated. An example is illustrated in the *Zero-Shot Code Translation Prompt* box. In this approach, the translation only relies on the model's *pre-training* knowledge to perform the task. However, due to LLMs *emergent abilities*, this method shows promising results and is used in multiple code translation publications [Yan+24b; Eni+24; Zho+25a].

Zero-Shot Code Translation Prompt

Instruction: Translate the following C code to Rust.

Code:

```
int add(int a, int b) {
    return a + b;
}
```

Few-Shot code translation prompting is similar to *Zero-Shot*, but extends the prompt with positive translation examples. The intuition behind this method is that the LLM learns the code translation in context and understands the desired output style [Li+24a; DSL25]. An example for *Few-Shot* prompting is illustrated in the *Few-Shot Code Translation Prompt* box.

However, since LLMs are known to be highly sensitive to prompts and structure, it is not a trivial decision what examples to incorporate in this prompting technique. Du et al. [DSL25] recently proposed a methodology addressing this problem. However, as it also impacts the number of processed tokens, the common prompt-based code translation relies on *Zero-Shot* [Yan+24b]. Therefore, the thesis referring to a *code translation prompt* implies a *Zero-Shot* code translation prompt.

Few-Shot Code Translation Prompt

Instruction: Translate the following C code into Rust.
Here is an example:
C Code:

```
int is_even(int n) {
    if (n % 2 == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Rust Translation:

```
fn is_even(n: i32) -> bool {
    n % 2 == 0
}
```

Now translate the following C code into Rust:

```
int add(int a, int b) {
    return a + b;
}
```

Fine-Tuning

Another approach for LLM-based code translation is *fine-tuning* a model specifically tailored to code translation. This approach utilizes the parameter weights learned during *pre-training* and employs a *fine-tuning* strategy to improve them for the specific use case of code translation. There are various methods for this approach, which are presented below. For a detailed explanation, however, please refer to the original papers. A common technique is leveraging *supervised fine-tuning (SFT)* with a specific training dataset containing ground truth translation pairs [Jan+24]. Other approaches use *unsupervised fine-tuning* or *self-supervised fine-tuning*, which train the model to reconstruct the original code from a corrupted version [Roz+20]. Furthermore, *reinforcement learning from compiler output and test execution* is used to improve the model weights [Jan+24].

Conclusion

While *fine-tuning* offers a higher specialization and may produce a higher accuracy, it requires specific data and computation, resulting in higher cost. When considering the rapid pace at which new and larger LLMs are published, *fine-tuning* is rather inflexible and fails to leverage this rapid progress, as it always requires a costly training before new models can be tested. Therefore, the thesis later evaluates a prompt-based code translation system.

2.4.2 Verifying LLM Translation

As stated earlier, LLM-based code generation must not be correct. It can, for instance, lead to incorrect logic [Din+23] or hallucinated methods [Tam+25]. Considering this, only verifying a translation by checking for *compilability* is not enough and is only a weak indicator for specifying a correct translation. A better indicator is the direct verification of *functional equivalence*, which is the ultimate goal of any code translation. Other LLMs for SE scenarios predominantly verify the correctness of a generation by unit tests. However, this approach requires comprehensive test suites, which are often unavailable for legacy codebases [Eni+24]. The code translation task comes with the benefit of producing an equivalent code and not a newly generated algorithm that solves a specific problem. Therefore, it can leverage a more flexible approach that automatically verifies the input and output equivalence of a generated code snippet, i.e., a *generate-and-check* pattern.

The *generate-and-check* pattern enables the opportunity to apply auto-repairing *feedback loops*, since failing translations are immediately recognized and can be re-prompted to the model. Therefore, such a *generate-and-check* pattern results in both a lower likelihood of an erroneous translation being undetected and the opportunity to leverage the pattern for a *feedback-approach*, which re-prompts the model in case of failure.

This section describes translation verification mechanisms that have been used in other SOTA LLM-based code translation systems focusing on *C* to *Rust*, in particular *FLUORINE* [Eni+24] and *VERT* [Yan+24b].

Differential Fuzzing

Eniser et al. employ *differential fuzzing* to directly verify translation output in their code translation system *FLUORINE* [Eni+24]. The general functionality of a *differential fuzzer* is to validate that two programs yield equivalent outputs for the same "randomly" fuzzed inputs. Eniser et al. name multiple publications that utilize *differential fuzzing* to check for *functional equivalence* [Pet+17; Guo+18; NNP20]. However, they state that *cross-language differential fuzzing* is a different task and is only sparsely found in other literature. *Cross-language differential fuzzing* requires executing programs written in

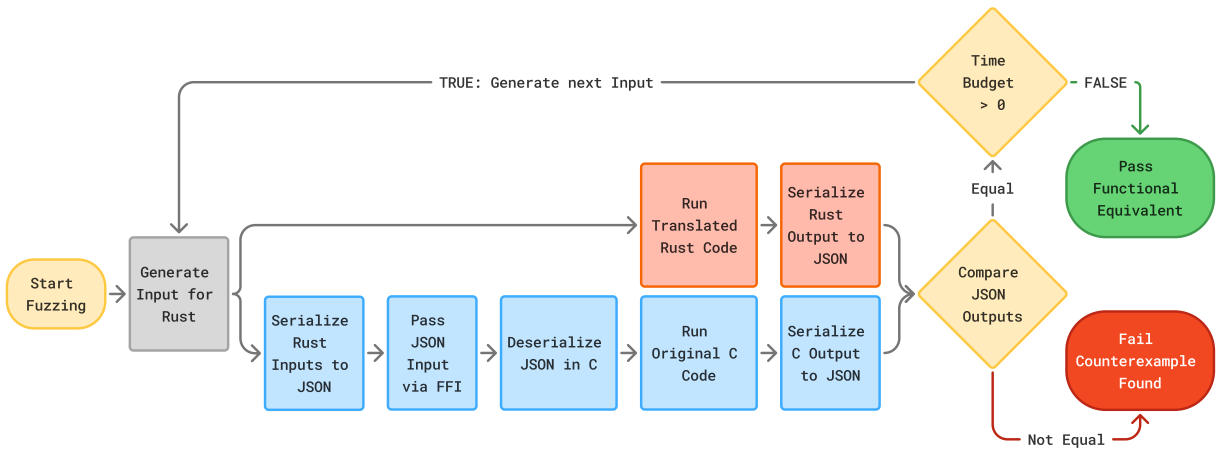


Figure 2.3: Abstract flow-chart visualization of the differential fuzzing process of *FLUORINE* [Eni+24].

different languages, with semantically equivalent inputs, and subsequently comparing their differently structured outputs.

With *FLUORINE*, they propose a detailed implementation that addresses this challenge. The core of their implementation lies in serialization and deserialization into *JavaScript Object Notation* (JSON) [Bra17] and an *Foreign Function Interface* (FFI). Such an FFI is an “abstraction used for interfacing a programming language with another foreign language to reuse its libraries” [Chi19].

FLUORINE’s differential fuzzing is visualized in Figure 2.3 and works as follows: It first utilizes the fuzzing framework *bolero*, which is based on *libfuzzer* [Ser16], and generates type-correct random inputs for the translated *Rust* function. These *Rust* input states are then serialized into a JSON representation, which is subsequently passed to *C* via FFI. With this mechanism, the *Rust* implementation is executed with the direct inputs generated by the fuzzer, whereas the original *C* implementation is called via FFI with inputs derived by de-serializing the equivalent JSON representation.

To compare the equivalence of their outputs, *FLUORINE* utilizes the same approach. The output of the *Rust* function is directly serialized to JSON and the original *C* function is serialized into JSON via FFI, whose *String* representation can then be “directly compared” for equivalence [Eni+24]. If the JSON strings of both languages are identical, the test case is considered passed. This leads to the fuzzer generating a new test case, as long as a specified *time budget* is not exhausted for the *differential fuzzing*. In case the JSON files differ, the *differential fuzzer* has identified a *counterexample* that indicates a semantic mismatch between the original *C* code and the LLM’s translated *Rust* code. If there are no *counterexamples* identified after the *time budget* is exhausted, the two functions are considered *functionally equivalent*.

This mechanism enables the desired automatic verification for the *generate-and-check* pattern, without relying on unit tests. However, Eniser et al [Eni+24] also name limitations of this process. The bottleneck is the completeness of the serialization and deserialization of complex and language-specific data types. Functions can only be fuzzed when they are serializable. Moreover, while the inherent *fuzzing* mechanism gives a high confidence in the equivalence results, it cannot give formal guarantees. Despite these limitations, *differential fuzzing* presents a valuable and highly autonomous process that can be leveraged for a *generate-and-check* approach. The completeness problem can be guarded by analyzing whether fuzzing failures are based on *counterexamples* or due to incompatible functions, which makes the bottleneck less drastic for a robustness evaluation, only focusing on performance deviations.

Formal Methods

Besides the *differential fuzzing* approach, there also exist *formal methods* that have the objective to mathematically prove equivalence, instead of validating it with testing. Consequently, this approach yields a stronger correctness guarantee than *differential fuzzing*.

Such an approach is applied in Yang. et al.'s *VERT*, an LLM-based *C-to-Rust* translation system. Their process starts by transpiling the original *C* code into a *WebAssembly (Wasm)* [Web] representation. Therefore, this step is basically a rule-based translation. Since this rule-based transpilation is only used for the functional equivalence proof, the drawbacks of rule-based transpilers, namely unidiomatic and unsafe code, do not negatively impact this process. The *Wasm* representation is then translated into *Rust* by embedding the *Wasm* semantics in *Rust* code with *rWasm*, a proven method to translate *Wasm* into *Rust* keeping semantic equivalence [BLP22]. The resulting *Rust* code represents an unreadable, but guaranteed functionally equivalent representation of the original *C* code, that later serves as "oracle" [Yan+24b].

To verify the readable LLM-based *Rust* translation, *VERT* utilizes two verification tools that compare the *oracle* and the LLM's translation. The first is *Kani*, a *bounded model checker* that automatically compares the behavior of two *Rust* programs [Van+22]. Since the authors encountered multiple cases where *Kani* was not able to verify equivalence and resulted in *timeouts*, they also applied an *autoactive verifier*, *Versus* [Lat+23]. This tool tries to mathematically prove equivalence based on annotations in the *Rust* code. Since this verifier relies on manual annotations, it is not fully autonomous.

While the process has a higher correctness guarantee than *differential fuzzing*, the not fully autonomous nature restricts the practical application. Consequently, with the thesis's objective of a comprehensive robustness evaluation framework, *differential fuzzing* is the better choice, enabling a *generate-and-check* pattern without relying on manual refinements. Therefore *FLUORINE's differential fuzzing* approach forms the basis for verifying translation correctness, which is explained in Chapter 4.

Iterative Refinement through Feedback Loops

Since the *generate-and-check* pattern reveals that LLMs do not always produce *compilable* or *functionally equivalent* translations, modern code translation systems like *FLUORINE* or *VERT* leverage the pattern for an auto-repairing, iterative refinement process in case of failure [Eni+24; Yan+24b].

The basic mechanism behind such a *feedback-approach* can be described as follows: The LLM initially generates a translation (*iteration one*). This translation is then automatically verified for successful translation. In case the translation is not valid, the system extracts the reason for the failure (i.e., the *feedback*). Specifically, such failure reasons can be *compiler* or *linting* errors, which are identified in both *FLUORINE* [Eni+24] and *VERT* [Yan+24b]. The *compiler* and *linter* give detailed descriptions, code-position, and suggestions for correction, which can be directly leveraged as *feedback*. Moreover, the *differential fuzzing* or *formal verification counterexamples* can be used to define *feedback* [Eni+24; Yan+24b].

This extracted *feedback* is then used to create a new LLM prompt, with the objective that the LLM repairs its erroneous translation. The papers use different strategies to refine the LLM prompt. *VERT* adds the *counterexamples* as test cases and therefore produces an example similar to a *Few-Shot* prompt. *FLUORINE* tries three different repair approaches: *(i)* a simple restart, without actually utilizing the feedback, *(ii)* a basic repair similar to *VERT's* approach, and *(iii) conversational repair*. The last approach is similar to *ii*, but does not remove *counterexamples* of the previous iteration.

In summary, the used refinement strategies are similar to *Few-Shot* prompting. However, the real *Few-Shot* prompting includes *correct* examples to improve the model's capabilities, whereas the refinement adds *incorrect* examples to the LLM's context.

VERT and *FLUORINE* point out that utilizing *feedback loops* increases the probability for a correct translation and is therefore a capability enabled by the *generate-and-check* pattern that can be leveraged for improved translation performance. However, while the refinement clearly improves the translation success, the authors have not explored whether such a strategy impacts the *robustness* of the LLMs. Thus this is reflected and assessed with RQ3: "Does incorporating a *feedback loop* strategy impact robustness?"

3 Related Work

Chapter 2 established the foundations of LLM-based code translation and detailed its potential, as well as the challenges that have to be considered. The explained *generate-and-check* pattern, with *differential fuzzing* and *feedback loops*, has the goal to improve the correctness of such an AI-based translation system. However, the stochastic nature inherent in their probability-based output generation raises a different question: the robustness of these systems. How reliable are these systems when they encounter minor changes in the input? Are they Sáenz's "*benefactor*", or do they act like a "*public enemy*"?

Since this examination is the main goal of this thesis, this chapter establishes the fundamentals for evaluating *robustness*. This includes a detailed definition of what is meant by the term robustness of an LLM-based system in Section 3.1. Furthermore, the chapter presents other robustness evaluation frameworks, relevant in today's literature (Section 3.2). Since these frameworks always utilize a modification of inputs, subsequently described as *perturbation*, Section 3.3 shows sources for perturbation strategies in the literature that can be leveraged for the proposed robustness evaluation framework of this thesis. Lastly, Section 3.4 summarizes the relevant work, shows research gaps, and identifies the necessary capabilities of the thesis's framework to address these research gaps.

3.1 How to Define Robustness

Before detailing related work in the area of robustness evaluation, it is necessary to define *robustness* at first. A common definition for *robustness* is the "degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions", which stems from *ISO/IEC/IEEE 24765:2017* a norm that "provides a common vocabulary applicable to all systems and software engineering work" [ISO17]. However, considering the nondeterministic fluctuations of LLM-based systems, this definition is not perfect. Consequently, there are various other and newer definitions of robustness that involve AI-based systems. The following enumeration lists available definitions and concludes afterwards which fits best to the robustness of an LLM-based code translation system.

- *ISO/IEC 22989:2022 Artificial intelligence concepts and terminology*: "ability of a system to maintain its level of performance under any circumstance" [IC22]
- *ISO/IEC TR24029-1 Assessment of the robustness of neural networks*: "ability of an AI system to maintain its level of performance under any circumstances" [IC21]
- *IEEE Std 3168 Standard for Robustness Evaluation*: "robustness of natural language processing (NLP) service: Capability of a natural language processing (NLP) service to maintain its level of performance in processing corrupted texts and adversarial texts." [IEE24]

- *ISO/PAS 8800 Safety and artificial intelligence*: “AI robustness: ability to maintain an acceptable level of performance under the presence of semantically insignificant but reasonably expected changes to the input” [ISO24]

Even though the definitions are not identical, they describe similar goals. A system is considered robust when it maintains its expected level of performance under any circumstances, such as invalid inputs, adversarial attacks, or *semantically insignificant but reasonably expected changes to the input*. Specifically, the definitions mostly contain the terms “its level of performance” or “acceptable level of performance”, which in contrast to *ISO/IEC/IEEE 24765:2017's* “function correctly” leave room for the inherent nondeterministic fluctuations of AI-based systems.

In detail, this thesis seeks to evaluate the robustness of an LLM-based code translation system and therefore assesses a generative AI for a specific practical real-world use case. Since this assessment only considers real-world usage, there is no need to evaluate a model's performance on *adversarial inputs*. *Adversarial attacks* typically try to maximize a model's loss function, to heuristically produce the most misleading inputs [Sze+14]. A developer or company that uses a code translation system has no motivation to attack the model with adversarial inputs. However, they could try to translate inputs with accidental typos, formatting changes, refactored and differently structured code, which could cause unforeseen performance deviations that do not adhere to “its acceptable level of performance”. Summarizing these definitions in light of the robustness of LLM-based code translation, which leads to distinguishing between Sáenz's *“benefactor”* or *“enemy”*, the definition considered in this thesis is as follows:

Robustness of LLM-based Code Translation

“An LLM-based code translation system is considered *robust* if it is able to maintain an *acceptable and expected level of performance* when prompted with semantically insignificant but reasonably expected changes to the input.”

3.2 Robustness Evaluation

While Chapter 2 presented the necessary fundamentals for going forward in this thesis, it is additionally necessary to display other publications focusing on a robustness evaluation in LLM-based code generation. This section presents the relevant works in this area. While robustness has also been studied in other contexts, such as NLP [MK22; Oma+22; Ahu+24], *image recognition* [Mod22; MBK23], and classical *machine learning* tasks primarily operating on tabular or time series data [Sie+23], these areas are out of scope for this thesis. This section specifically focuses on robustness in the context of LLM-based *code generation*, which enables the identification of research gaps in Section 3.4 and additionally gives inspiration for the development of a framework that closes these gaps.

3.2.1 ReCode: Robustness Evaluation of Code Generation Models

“ReCode: Robustness Evaluation of Code Generation Models” [Wan+23] presents a benchmark specifically designed to assess the robustness of code generation models. It introduces a robustness evaluation approach for code generation models that incorporates metrics

capturing worst-case performance across perturbed inputs. Although the paper focuses on docstring-based code generation and plain code completion, its underlying methodology can be adapted to address the research questions of this thesis. Therefore, this particular publication is described in high detail.

Approach

ReCode is build on existing code generation benchmarks such as *HumanEval* [Che+21], and MBPP [Aus+21] (see Section 2.3.1). These benchmarks require the generation of code either from docstrings or via problem-specific prompts, which are subsequently verified using unit tests. That means, for each sample (either docstring or problem-specific prompt), there exist unit tests to verify the correctness of the LLM's output.

To leverage these datasets for a robustness evaluation, the core idea of *ReCode* is to systematically perturb input prompts using various strategies that aim to preserve semantic meaning and naturalness, to get results that are likely to appear in practice. For a more detailed assessment, the paper differentiates perturbations into four categories, based on the target the perturbation aims to create change: *docstrings* (10 strategies), *function names* (6 strategies), *code syntax* (6 strategies) or *code format* (6 strategies).

Given semantically similar yet structurally different prompts enables the evaluation and comparison of a model's performance across diverse perturbation strategies, which can ultimately be used as a measure of robustness. Specifically, the *pass@k* results for the various perturbation strategies can be used to determine whether, and to what extent, particular strategies affect the success on the unit tests (*functional correctness*).

ReCode utilizes multiple stochastically generated perturbation strategies. More precisely, this refers to perturbations that can result in different modifications for an identical original input. Since each stochastic variation can potentially affect the LLM differently, *ReCode* creates multiple samples for each perturbation. In detail, *ReCode* creates s randomly perturbed prompts and measures the worst-case performance across each group of s perturbed prompts for a single perturbation strategy. In their paper, a model is considered robust for a specific perturbation if and only if it generates a correct solution for all s perturbed prompts, which the authors describe as a worst-case approach.

Robustness Evaluation Metrics

In order to quantify their definition of robustness, which incorporates the different variations per prompt, *ReCode* introduces three novel metrics for robustness evaluation.

Robust Pass$_s$@k (RP$_s$@k) *Robust Pass$_s$@k* (RP$_s$@k) extends the standard *pass@k* metric by inlcuding the idea of a worst-case performance measure across s randomly perturbed prompts.

For an original prompt x[1], a single perturbation strategy is applied to create s perturbed prompts, i.e., $x_1, \dots, x_s$. For each perturbed prompt x_j, the model generates n solutions, leading to a total of $n \cdot s$ generated solutions $f_i(x_j)$, where $1 \leq i \leq n$ and $1 \leq j \leq s$. This metric specifically aligns to their worst-case robustness definition: a task is considered generated correctly under a specific perturbation only if it is correct for all s perturbations

[1]Also described as a task to avoid confusion when working with perturbed prompts and multiple n solutions.

of the same perturbation strategy, where correctness is quantified by passing unit tests. According to this, a correct solution is defined as:

$$c_{i,s}(x) = \begin{cases} 1, & \text{if } f_i(x_1), \dots, f_i(x_s) \text{ are all correct,} \\ 0, & \text{otherwise.} \end{cases}$$

The number of robustly correct solutions is given by summing over all worst-case aggregated outputs:

$$rc_s(x) = \sum_{i=1}^{n} c_{i,s}(x).$$

Following the standard *pass@k* definition, *RobustPass$_s$@k* is defined as Equation (3.1):

$$\mathrm{RP}_s@\mathrm{k} = \mathbb{E}_x \left[1 - \frac{\binom{n - rc_s(x)}{k}}{\binom{n}{k}} \right]. \tag{3.1}$$

Thus, RP$_s$@k represents the probability that, for a randomly chosen task x, at least one of the k randomly selected outputs is robustly correct, i.e. correct for all s perturbed prompts. Therefore, higher values of RP$_s$@k indicate a greater likelihood of consistently robust performance.

Robust Drop$_s$@k (RD$_s$@k) *Robust Drop$_s$@k* (RD$_s$@k) measures the relative performance degradation of a model under perturbed prompts compared to its original performance. Given an original prompt x and its perturbed versions $x_1, \dots, x_s$ for a single perturbation strategy, RD$_s$@k is defined as Equation (3.2):

$$\mathrm{RD}_s@\mathrm{k} = \frac{\mathrm{pass@k} - RP_s@k}{\mathrm{pass@k}}. \tag{3.2}$$

Higher RD$_s$@k values signal greater sensitivity to input variations leading to performance variations, whereas values closer to zero suggest robust behavior, with few performance deviations under the perturbation.

Beyond these metrics, *ReCode* introduces a third metric, namely *Robust Relative$_s$@k* (RR$_s$@k). However, its underlying concept is not relevant for this thesis. Specifically, the metric accounts for single perturbed solutions deviating from their unperturbed counterpart, in both negative and positive deviations. During the development and testing of the thesis's framework, RR$_s$@k was perceived as unintuitive and not necessary to include, as RP$_s$@k also indicates whether a model improves or worsens under a perturbation. More information and the detailed definition of this metric can be found in the original paper of *ReCode* [Wan+23].

Validating Perturbations

A key requirement of *ReCode*'s core idea is that the perturbed prompts must be natural and preserve semantic meaning. Outputs of semantically different prompts would most certainly fail the unit tests, resulting in worse model performance and hence would be assessed as less robust. Additionally, using perturbed prompts that are not likely to appear in practice is excluded. The goal of *ReCode* is to assess an LLM under real-world conditions, and unnatural prompts are not part of this.

To verify that this requirement is fulfilled, the paper uses two automatic and one human-based strategy, where the human-based approach is described as generally more reliable. For human evaluation, five human annotators are asked to rate naturalness and semantics for a random sample of prompts. This is a tedious and time-consuming approach, especially for very large datasets, which motivated the exploration of automatic strategies. The two automatic approaches use the metrics cosine similarity and *CodeBLEU* [Lu+21; Ren+20]. *Cosine similarity*, described in Section 2.2.1, is used to measure the sentence similarity based on encoded embedding vectors for *docstring* and *function name* perturbations. The remaining perturbation categories: *code syntax* and *code format*, are verified using *CodeBLEU* scores. *CodeBLEU* [Lu+21; Ren+20] is a metric that is normally used to compare automatically generated code to a reference solution. It is an adaptation of the well-established *BLEU* [Pap+02] metric, which has its origins in machine translation, but additionally adds aspects for improved functionality when working with code. *ReCode* assesses the similarity between the perturbation and the original prompt, using the original prompt as the reference solution.

Evaluation Results

ReCode involves more than 28 perturbation strategies that were validated for naturalness and semantic similarity. The publication presents a *ReCode*-based robustness evaluation for three different models: *CodeGen* [Nij+23], *InCoder* [Fri+23], and *GPT-J* [Wan21]. During the time this paper was published, these models represented SOTA for code generation. However, it should be noted that modern LLM significantly outperform them in terms of functional correctness on SOTA code generation benchmarks [Zhe+23]. In addition, their experimental setup evaluated the models with *greedy-sampling*-based output generation, which has been shown to hinder the model's capabilities [Hol+20] and does not fully reflect how LLMs are normally configured. Yet this enabled the evaluation to reduce the problem of nondeterministic noise.

By applying *ReCode's* methodology on the models, the authors describe multiple noteworthy robustness findings. First, they discover that a more diversely pretrained model like *CodeGen* [Nij+23] results in higher $\text{RP}_s@\text{k}$ values, but with more susceptibility to relative performance drops under perturbations in $\text{RD}_s@\text{k}$. Secondly, they observe the same behavior when comparing different model sizes of *CodeGen*. Larger models show higher $\text{RP}_s@\text{k}$, but simultaneously increase the relative performance drops $\text{RD}_s@\text{k}$. Another finding is that the LLMs signal the strongest weaknesses for syntax perturbations, which, in light of the code-focused variations in this thesis, is particularly relevant. Lastly, since they evaluate the models on two different datasets, namely *HumanEval* and MBPP, they discover that the LLMs robustness is worse on MBPP. They explain that this is due to MBPP containing more diverse variations in code style, consequently better reflecting the diverse nature of real-world codebases.

3.2.2 Other Robustness Evaluation Approaches

Although *ReCode* [Wan+23] represents the most relevant work for this thesis, other robustness evaluation approaches are also noteworthy.

Robustness of GitHub Copilot against Paraphrased Instructions

Mastropaolo et al. [Mas+23] examined the robustness of *GitHub Copilot* by analyzing how perturbations on natural language instructions affect code generation. At the time of their work, *GitHub Copilot* was based on *OpenAI*'s *Codex* [Che+21]. A code LLM fine-tuned on *GPT-3* [Bro+20] for docstring-based code completion. Mastropaolo et al. measured robustness by comparing the similarity of generated outputs using token-level *Levenshtein distance* [Lev66] and *CodeBLEU* [Lu+21; Ren+20], complemented by assessing correctness based on unit test results.

The publication's evaluated use case was function generation given *Javadoc* [Orab] descriptions and a method signature. Similarly to *ReCode*, robustness was assessed by perturbing the inputs. Specifically, the approach of the paper was to perturb the *Javadoc* descriptions using either a *deep learning*-based paraphrasing model [Zha+20], a back-translation strategy[2], which translates the text into a different language and reverts it by translating it back again, or by manual paraphrasing. The model was subsequently prompted with both the original and the perturbed descriptions, and its outputs were analyzed with the three methods described above.

Their results show that 46% of the paraphrased inputs led to a structurally different code generation. Moreover, they found that 28% of the functionally correct solutions were unique to either the original or perturbed input, consequently signaling non-robust behavior. Beyond that, Mastropaolo et al. find that higher output similarity positively correlated with functional correctness. They conclude that larger structural differences often lead to incorrect completions.

This work shows that even slight variations to a prompt can lead to worse model performance and different results, which underlines the importance of a comprehensive robustness assessment of SOTA-LLMs. However, this approach only focuses on natural language instructions, for which LLMs potentially can have a different sensitivity than for variations of code. In addition, compared to *ReCode*, this evaluation only assesses a small number of perturbation strategies. Also, it does not accurately account for the nondeterministic nature of LLMs. In theory, the model could generate structurally different outputs for the same prompt. While it is interesting to observe that the model produces structurally different results for similar prompts, the practical relevance of this observation depends on the specificity of the instruction. One could argue that different programmers would also write structurally different functions, given the same task. As long as a result is idiomatic and functionally correct, structural variation should be considered acceptable. Neither of which can be measured by comparing the structural similarity between outputs of different prompts. Furthermore, while the paper demonstrates a correlation between output similarity and functional correctness, it does not investigate whether a stronger magnitude of perturbations led to worse functional correctness.

COCO: Testing Code Generation Systems via Concretized Instructions

"*COCO: Robustness testing of code generation systems via concretizing instructions*", introduces another approach for the robustness assessment of code generation models [Yan+23a]. Similar to Mastropaolo et al. [Mas+23], *COCO* perturbs natural language instructions. However, according to the authors, their novel strategy causes significantly stronger robustness deficiencies than Mastropaolo et al.'s approach.

[2]The authors defined this process as Translation Pivoting.

COCO's perturbation extracts code features, such as `loops`, `function definitions`, or `library imports`, out of the generated solutions when prompted with the original instruction. The identified features are then used to create a more concretized version of the original instruction. In detail, the perturbed instruction extends the original instruction by specifying what code features are required and what features are absent. So *COCO*'s strategy is to automatically add requirements or guardrails to the prompt, which by definition results in a semantically similar task. Similar to *ReCode* and Mastropaolo et al., this enables the comparison of unperturbed and perturbed prompts. However, instead of directly comparing the unit test pass rate against the original prompt, *COCO* measures robustness by counting three types of "inconsistencies". Specifically, they check *(i)* whether a model succeeds on the original instruction but fails unit tests under the perturbed one, *(ii)* whether a model produces compilable code on the original instruction but fails under the perturbed one, and *(iii)* whether a generated code feature is present for the original instruction, but absent under the perturbation. The more inconsistencies a model produces, the stronger the robustness deficits of a model.

COCO has been applied to multiple code generation models on previously described benchmarks, i.e., *HumanEval* and APPS (Section 2.3.1). A noteworthy finding is that increasing the number of appended concretized instructions led to stronger robustness deficits.[3] By that, it is the first robustness evaluation that empirically shows that the magnitude of the perturbation increases robustness deficits. What raises the question of whether this behavior is specific to their perturbation strategy or a more general phenomenon in robustness evaluation.

Enhancing Robustness of AI Offensive Code Generators via Data Augmentation

Besides *COCO* and the work by Mastropaolo et al., a more recent paper focuses on the robustness of AI offensive code generators[4] [Imp+25]. Again, this evaluation applies perturbations on natural language instructions and compares unperturbed outputs with perturbed outputs. In detail, the paper utilizes two perturbation strategies: *word substitution* and *word omission*. *Word substitution* replaces a word with a semantically similar one, whereas *word omission* randomly removes a word from the prompt.

The paper utilizes three metrics: *Syntactic Accuracy*, *Semantic Accuracy*, and *Robust Accuracy* to measure robustness. *Syntactic Accuracy* quantifies the accuracy of compilable code. *Semantic Accuracy* measures whether the output is semantically correct. However, compared to other approaches, the authors manually judged whether the generated code was semantically correct. Lastly, *Robust Accuracy*, introduced by Huang et al. [Hua+21][5], can be described as RD_s@k configured with $n = k = s = 1$. Applying the perturbations, the authors discover non-robust behavior, particularly when information from the prompt is removed by *word omission*.

Similar to *ReCode*, this work incorporates *cosine similarity* to check whether the perturbations preserve the original meaning. Perturbed prompts that yield a *cosine similarity* lower than a predefined threshold are being excluded from the robustness evaluation. It should be noted that the authors also validated this *cosine similarity* approach. To be more precise, they asked 31 human annotators to assess whether the perturbed prompt is semantically similar to the original instruction. It showed that *cosine*

[3]This behavior can be controlled via *COCO*'s hyperparameters.

[4]This task describes generating code for offensive security activities like penetration testing.

[5]A work that evaluates the robustness of AI models converting natural language into machine-readable formats.

similarity is a feasible approach to exclude misleading perturbations. Although the authors admit that *cosine similarity* is not a perfect solution, they conclude that *cosine similarity* is an approach that sufficiently evaluates the effects of perturbations. Considering that the thesis seeks to incorporate the semantic similarity of perturbations into the evaluation, this is particularly relevant. To make this framework dynamically applicable to different datasets, it is necessary to minimize manual interaction.

Adversarial Robustness Evaluation

Beyond the previously described publications, several works focus on adversarial robustness [Mic+19; Li+19; Che+20; WAV20; Bou+22]. That means instead of using well-intentioned perturbations, these works specifically design inputs that attempt to mislead the model by maximizing their loss function for a specific input. Although this is not always easily realizable, as many LLMs act as black-box models, obscuring their internal loss functions, approaches have still been developed to generate adversarial inputs. However, this thesis will not include adversarial perturbation strategies. The goal of this thesis is to evaluate the robustness of an LLM-based code translation system in a real-world, relevant context. That refers to a well-intentioned programmer utilizing the system for a code translation task. Such a programmer has no motivation to forcefully mislead the model to produce inaccurate translations, making adversarial robustness irrelevant in this context.

3.3 Perturbation Strategies

This previous section indicates that all related robustness evaluations share perturbations as a common element. Every approach inputs original and perturbed prompts into an LLM to compare differences in the model's outputs. While all papers apply different perturbation strategies, the predominant focus is on perturbations targeting natural language instructions.

When considering the evaluated use cases, this is reasonable, as instructions are the pivotal part of problem-based code generation or code completion, and will vary drastically in practical applications. However, in the context of code translation, such variations in the instruction are less relevant. A deployed code translation system will only change in the `source language` and `target language` of the translation, but not in the general instruction, e.g., "translate the following <`source language`> code into <`target language`>". However, the code to be translated may involve diverse sets of variations in practical applications. Therefore, this thesis particularly focuses on perturbations to code

Section 3.2 details that the goal of a perturbation strategy is to produce a transformation semantically equivalent to the original prompt. As far as is known, *ReCode* [Wan+23] is the only robustness evaluation on LLM-based code generation that incorporates code-focused perturbation strategies. However, most of its proposed perturbations act at a superficial level, such as renaming identifiers or applying natural language perturbations to docstrings. *ReCode* categorized perturbations into four groups: *docstrings*, *function names*, *code syntax*, and *code format*. While six perturbation strategies target the code's syntax, only three of them actually modify the code's control flow. The presented categories do not enable a direct assessment of whether perturbations operate on a superficial or profound level. This leaves the impression that deeper, more fundamental perturbations may be missing.

Hence, the thesis must investigate additional perturbations to the code. Fortunately, generating semantic-preserving transformations is not only relevant for robustness evaluations. The literature shows that similar techniques are applied in the areas of *Data Augmentation* for code, *Code Clone Detection*, and *Source Code Plagiarism Detection*. The following sections mention possible sources for perturbation strategies that can be referred to identify other automatic perturbations targeting code.

3.3.1 Data Augmentation

The goal of the *Data Augmentation* domain is to artificially expand a model's training dataset by transforming the existing data. This technique is well-known in computer vision [KSH12] and is now also adopted to extend code-based datasets. Some publications propose semantic-preserving perturbations that overlap with some of *ReCode*'s perturbations, such as converting for loops into while loops or renaming variables [Li+22; YWW22]. However, the literature also shows novel perturbations. For instance, a recent publication introduced **ConditionDup**, where logically neutral elements (e.g., && *True* or || *False*) are added to *conditional expressions* [Li+24b].

3.3.2 Code Clone Detection

Code Clone Detection seeks to identify semantically similar code snippets across a codebase to support clearing redundancy to improve maintainability and prevent bugs [Dou+23]. A common way to evaluate these methods is through code transformations, such as adding dead code snippets or removing comments [Zha+23b].

3.3.3 Source Code Plagiarism Detection

Source Code Plagiarism Detection has similarities with *Code Clone Detection*, but it particularly targets intentional modifications of existing code to obscure plagiarism. In order to be practically relevant, these systems must be robust against significant and not well-intentioned modifications. A programmer attempting to plagiarize code will likely go beyond renaming variables. The programmer might combine multiple transformation strategies to create functionally equivalent but syntactically distinct code.

To classify the magnitude of such functional equivalent modifications, Faidhi and Robinson [FR87] proposed a methodology consisting of six levels of program transformation. These levels go from superficial changes (e.g., comments and indentation) to more profound modifications to decision logic and control flow.

Figure 3.1 gives an illustration of the proposed levels and details that each higher level includes all modifications of the lower levels. Level I includes changes to comments and indentation. Level II extends Level I by including changes to identifiers, such as renaming variables, functions, or constants. Transformations in Level III involve changes to declarations, which means the introduction of unused constants, for instance. Level IV goes beyond that and includes modifications to functions, such as extracting code blocks into separate functions or changing method signatures. Level V is described as "transformation of semantic equivalents", which incorporates modifications that transform a while loop into an equivalent for loop and vice versa. Finally, Level VI includes the most profound modifications, including changes to decision logic and expressions, such as the **ConditionDup** [Li+24b] perturbation mentioned in Section 3.3.1).

These levels provided a basis for selecting perturbation strategies for *SPPlagiarise* [CLS19]. *SPPlagiarise* simulates *Java* code plagiarism by automatically applying perturbations

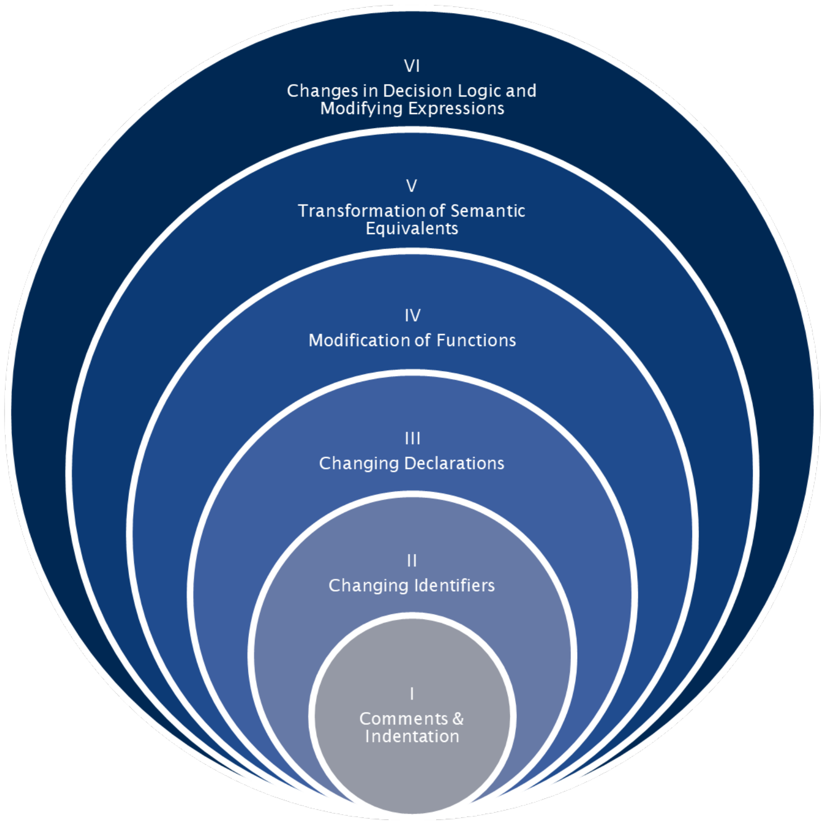

VI
Changes in Decision Logic and Modifying Expressions
V
Transformation of Semantic Equivalents
IV
Modification of Functions
III
Changing Declarations
II
Changing Identifiers
I
Comments & Indentation

Figure 3.1: Six levels of program transformations, inspired by [FR87]

from levels *I* to *V*. This paper introduced various Java-specific perturbations that go beyond simple identifier renaming. Consequently, their work inspired a great number of the strategies explored in this thesis. The particular perturbation strategies that have been used for this thesis will be explained in the next chapter. However, before presenting the methodology of this thesis, it is necessary to give an overview of the related work and show the unexplored gaps, which should be subsequently solved by the proposed methodology.

3.4 Discussion of Related Work in Context of the Research Gaps

When comparing the relevant related works in Table 3.1, it turns out that there are multiple strategies to assess a model's robustness in code generation. While they are all interesting in themselves, there remain unexplored research gaps. The thesis's introduction specifically mentioned these research gaps: *(i)* robustness to instructions vs robustness to code, *(ii)* robustness when translating code, *(iii)* programming languages and benchmark complexity, *(iv)* model modernity, and *(v)* unexplored aspects of robustness. This section discusses the different works in the context of the research gaps and defines the goals for the thesis's evaluation framework.

3.4.1 Robustness to Instruction vs Robustness to Code

Table 3.1 demonstrates that most papers perturb instructions, which makes sense given their respective use cases, where code is not always included in the LLM prompt. In

Paper	Use Case	Perturbation Target	Metric	Similarity	Model	Dataset	Language
ReCode [Wan+23]	Docstring, Task-based	Instructions, Comments, Code	RP_s@k, RD_s@k, RR_s@k	Cosine Similarity, CodeBLEU, Human Review	CodeGEN, InCoder, GPT-J	HumanEval, MBPP	Python
Mastropaolo et al. [Mas+23]	Docstring, Function Signature	Instructions	Structural Similarity, Functional Correctness	Levenshtein, CodeBLEU	Codex	Open-Source-Projects	Java
COCO [Yan+23a]	Task-based	Instructions	Inconsistency Count	CodeBLEU	GPT-3, GPT-3.5, CodeGEN, CodeRL, InCoder, PyCodeGPT, GPT-2, Code-T5, CodeGen, InCoder	HumanEval, APPS	Python
Improta et al. [Imp+25]	Task-based	Instructions	Syntactic Accuracy, Semantic Accuracy, RD@k	Cosine Similarity, Human Review	Seq2Seq, CodeBERT, CodeT5+	Exploit-db, Shell-storm	Assembly

Table 3.1: Comparison of relevant related work.

the context of LLM-based code translation, it is more interesting to investigate whether variations of the inputted code (such as formatting, variable names, and comments) cause non-robust behavior. While *ReCode* is a publication that includes both perturbation strategies on instructions[6] and on perturbation strategies on code, their perturbation only sparsely modify the code's control flow. Therefore, it lays a good foundation for the effects of perturbations on code, but could be extended to cover a broader range of code-focused perturbations. This thesis will adopt the core approach of *ReCode* and extend the number of perturbation strategies, shifting their focus to perturbations that are more relevant for code translation.

3.4.2 Robustness When Translating Code

As Table 3.1 shows, none of the existing studies specifically focus on code translation. Therefore, the robustness of LLMs in code translation is completely unexplored. Recalling Nezhurina et al. [Nez+24], an LLM's capabilities in different domains should not be directly transferred to other presumably similar domains, as this could lead to an overestimation of its capabilities. Therefore, a robustness evaluation specifically tailored to code translation is necessary.

3.4.3 Programming Languages and Benchmark Complexity

Table 3.1 details that robustness evaluation has mostly focused on popular languages like *Python* or *Java* [Cas24]. While Improta et al. assess the robustness for generating *Assembly*, the thesis's desired languages *C* and *Rust* are absent in the robustness evaluation literature. With *ReCode* being the most relevant paper, because of its perturbations to Code, it is worth discussing its evaluated benchmarks. Recall that *ReCode's* results suggested that

[6]To be more accurate, they include perturbation on *docstrings*, which in the context of the evaluated code completion are similar to the natural language instructions observed in code translation.

robustness is worse on the more complex MBPP (Section 3.2.1). However, both their evaluated datasets are known to be simple, consisting of straightforward coding interview-style questions and do not fully reflect real-world programming scenarios [Aga+24; Sid+24; YBS24]. So they found robustness issues even on datasets with easier problems than the real world may offer in practical applications. The discussion of RQ5 will investigate if their discoveries can be transferred to the robustness of modern models, translating real-world relevant *C* code into *Rust*. Furthermore, it has been shown that these particular benchmarks are used in the training of modern SOTA LLMs [Mat+24]. As a result, performance on these benchmarks may overestimate a model's actual robustness when applied to unseen real-world code. This may not be true for the models evaluated by *ReCode*, but when considering the next research gap, the benchmark datasets are not a good choice for robustness evaluations that seek to be relevant today.

3.4.4 Model Modernity

Table 3.1 lists the evaluated models. A recent survey shows the benchmarking performance of different models [Zhe+23]. Referring to this comparison, it becomes clear that most of the evaluated models are now outdated. Zheng et al. show that larger and more recent models tend to achieve better results in code-related tasks. However, the models assessed in previous studies are primarily specialized for code generation (with the exception of *COCO*, which also evaluated earlier versions of *GPT*). Since many general-purpose models now outperform dedicated code generation models, it would be interesting to investigate whether they also have robustness issues. One could argue that, because of their larger general knowledge and improved natural language capabilities, they might be more resistant to natural language-based perturbations. Therefore, an updated robustness evaluation with modern models remains to be conducted.

3.4.5 Unexplored Nuances of Robustness

Section 1.2 mentioned nuances of robustness, such as *(i) feedback loops*' impact on robustness, *(ii)* distinguishing between nondeterministic fluctuations and true robustness, as well as *(iii)* the correlation between input similarity of perturbations and robustness.

Since no evaluated approach contains iterative repairs with *feedback loops*, *(i)* remains unanswered.

Regarding the nondeterministic fluctuations, *ReCode* and *COCO* are publications that considered this problem. *ReCode* performed a vast number of runs $n = 100, s = 10$ to reduce the effects of stochasticity. In addition, both *ReCode* and *COCO* used *greedy-sampling*-based output generation, which reduced the risk of misinterpreting fluctuations as non-robust behavior. However, since *greedy-sampling* is a rarely used output sampling method, their evaluation only bypassed this problem, without delivering a methodology that allows a distinction between noise and genuine robustness issues for a more common output sampling strategy.

Lastly, all studies validate their perturbations using similarity scores. However, none explicitly demonstrate how similarity correlates with robustness sensitivity. *COCO* is the only work that provides information that more concretized instructions result in reduced robustness. For all other perturbations, this has not been investigated. So it is unclear whether the reduced robustness is primarily driven by the magnitude of input changes or if it follows other patterns. One would expect a model to perform well on perturbations where changes are minor and to perform worse on strongly transformed prompts. Whether this intuition stands for perturbations on code remains to be seen. If such a correlation can be found, one could use the similarity to predict the robustness of the model.

3.4.6 Summary and Goals for the Thesis

Examining the existing relevant literature reveals that the general methodology for evaluating robustness is similar across approaches. The basic idea is always to produce prompt variations (perturbations) that are compared to the LLM's unperturbed performance. While this comparison is measured differently for approaches, the general concept of quantifying performance with correctness metrics is relevant for each of the presented approaches. So, the basic methodology for evaluating the robustness can be adopted from the existing work. However, the thesis's methodology has to account for the existing research gaps that have been shown and propose an approach specifically tailored to *C-to-Rust* code translation. That means the framework has to include a large range of differently complex perturbations targeting code. Some perturbation ideas can be taken from *ReCode* and other perturbation-based areas. The six levels of Faidhi and Robinson [FR87] visualized in Figure 3.1, can additionally be used to classify the perturbation strategies to ensure that the framework incorporates perturbations of different complexities.

Furthermore, the existing works either ignored the problem of nondeterminism or employed *greedy-sampling*. Therefore, the literature misses a strategy for evaluating the robustness of LLMs when configured with the more relevant temperature-based output sampling strategy. The thesis will provide a strategy to identify non-robust behavior even for the nondeterministic temperature-based sampling.

Moreover, the methodology will enable the exploration of the nuances of robustness. That means it will measure the robustness with and without *feedback loops* to enable the discussion of whether *feedback loops* impact the robustness. Similarly, the methodology will incorporate the semantic similarity aspect into conclusions about the robustness of a perturbation.

Lastly, to investigate whether robustness results are model-specific or model-agnostic, the framework will enable the usage of different LLMs, which additionally can be used to address the problem of *model modernity*.

By developing a framework that addresses these goals and by applying it on a real-world relevant *C*-dataset, the thesis can explore the RQs that arise due to the identified research gaps.

4 Methodology - The Framework

This chapter details the proposed robustness evaluation framework, which is specifically tailored to LLM-based *C-to-Rust* translation. The earlier chapters show that, despite the increasing use of LLMs in software engineering, their robustness to real-world variations in prompted code remains largely unexplored, especially when translating code.

To explore the research gaps and deliver empirical information to answer the RQs, the thesis introduces a three-step framework. This framework is based on the basic idea of prior robustness evaluation works. This means it compares the regular performance of the system against the performance when prompted with input variations. Therefore, the entire methodological approach of this thesis is based on the premise that a perfectly robust model should be invariant to semantically equivalent prompt variations. To be more precise, a robust translation system should not degrade in translation success when faced with irrelevant, superficial, or profound prompt variations. Since the related work missed out on crucial research gaps, the basic methodological approach is extended with novel aspects to explore the research gaps. Specifically, each component of the framework contributes to answering the RQs. By demonstrating what is necessary for a comprehensive robustness evaluation, the overall framework design, including its components, addresses RQ1. RQ2 asks "How does one differentiate between inherent LLM nondeterminism (noise) and true robustness deficits?"

While this is not directly incorporated as a component, the thesis proposes a methodology for application of the framework which helps to differentiate between model nondeterminism and true robustness deficits (Section 4.5). Furthermore, the framework involves a *feedback loop* approach for RQ3 and also includes a semantic similarity analysis to account for RQ4. Lastly, the framework's model-agnostic design allows a comparison of robustness across different LLMs (RQ5).

By systematically applying this framework and analyzing the resulting data, this thesis provides the necessary evidence to address all research questions comprehensively.

4.1 Overview of the Framework

Figure 4.1 illustrates that the framework follows three separate steps. In the beginning, it requires a benchmark dataset consisting of multiple *C* code files as input. Recalling the research gap "programming languages and benchmark complexity", the framework directly enables choosing a more complex and real-world relevant dataset, which will be discussed in Chapter 5. The dataset's *C* files are then processed by the framework to produce prompt perturbations in *Step I* and translate them into *Rust* in *Step II* to produce the translation statistics required for the robustness evaluation in *Step III*.

Step I - Perturbation

In the first step, the framework generates multiple perturbation datasets based on a *C* to *Rust* code translation benchmark dataset. As previous research proved, perturbations

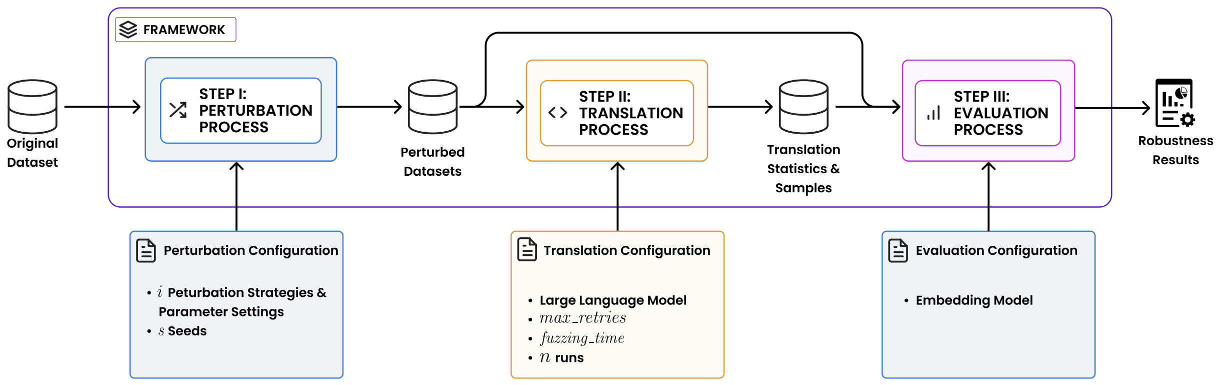

FRAMEWORK
Original Dataset
STEP I: PERTURBATION PROCESS
Perturbed Datasets
STEP II: TRANSLATION PROCESS
Translation Statistics & Samples
STEP III: EVALUATION PROCESS
Robustness Results
Perturbation Configuration
• i Peturbation Strategies & Parameter Settings
• s Seeds
Translation Configuration
• Large Language Model
• $max\_retries$
• $fuzzing\_time$
• n runs
Evaluation Configuration
• Embedding Model

Figure 4.1: Overview of the proposed three-step framework, consisting of a perturbation, translation, and evaluation process.

are a crucial part when evaluating the robustness of an LLM in the context of code generation. Utilizing variations of the original prompt enables the comparison of the model's performance regarding different perturbations against the original prompt.

Step II - Translation

Step II utilizes the perturbed datasets of Step I and calls an LLM with all the code translation prompts resulting from Step I. Step II aims to generate the *Rust* translations and record translation statistics, which will be used in Step III. In detail, these statistics capture whether a translation is successful, determined by two criteria: (*i*) *compilation success* (i.e., the *Rust* code compiles without errors) and (*ii*) *fuzzing success* (i.e., a *differential fuzzer* detects no behavioral differences between the original *C* code and its *Rust* translation, thereby testing their functional equivalence).

Step III - Evaluation

Lastly, Step III analyzes the translation results of Step II and evaluates them according to predefined metrics. Specifically, the metrics of the framework are based on previously mentioned *pass@k* approaches, namely RP_s@k introduced by Wang et al. in *ReCode* [Wan+23]. However, rather than using unit tests to measure correctness, the framework relies on the translation statistics from Step II (*compilation success* and *fuzzing success*), similar to other LLM-based code translation works [Eni+24].

In addition to the robustness metrics, the proposed evaluation step incorporates the concept of perturbation similarity into the robustness assessment. Consequently, Step III also receives the perturbed datasets of Step I as input and quantifies their similarity to the original prompt. To be more precise, Step III uses an embedding model to compute the *cosine similarity* between the original and the perturbed files. The intuition is that perturbations that are very similar to the original yield more robust translation results compared to less similar perturbations. Capturing this effect is a nuance of a comprehensive robustness evaluation and also accounts for RQ4. An LLM that shows non-robust behavior to highly similar input variations may be deemed less robust, as a model that primarily fails for perturbations that are significantly different from the original.

Furthermore, Figure 4.1 illustrates that, in addition to the benchmark dataset, the framework also requires a configuration for each step. The specific details of each step are explained in the upcoming sections.

4.2 Step I - Perturbation

In the first step, the framework systematically produces perturbed versions of the original *C* code files combined with a predefined code translation instruction. As Chapter 3 highlights, this is crucial when wanting to assess how an LLM handles variations in code or instructions, reflecting realistic programming styles or minor user mistakes like spelling.

Step I of the framework runs based on a configuration file. This configuration specifies which perturbation strategies to use and with what parameter settings. This results in the framework generating a perturbed dataset for each strategy. Specifically, the framework includes 23 distinct strategies that, through different parameter configurations, can lead to a wide range of perturbed datasets.

The following subsections address the main challenges of perturbation-based evaluation, introduce a categorization for improved distinguishability, and detail each of the 23 implemented strategies with examples. Finally, the overall perturbation process and implementation details are outlined.

4.2.1 Challenges and Perturbation Requirements

In order to comprehensively assess and compare an LLM's robustness in code translation, multiple perturbation strategies are required. Recalling the "robustness to instruction vs robustness to code" research gap (Section 1.2.1), a robustness evaluation relevant to code translation has to primarily check for input variations in code. Specifically, a robustness evaluation should analyze how much the model is affected by real-world noise and inconsistencies in its prompted code. To evaluate this, the perturbations have to reflect such behavior accurately. However, identifying and implementing such perturbations presents several challenges.

Semantic Equivalence

A fundamental requirement for robustness evaluation is ensuring semantic equivalence across different perturbation strategies. As seen in prior work, most robustness assessments rely on semantically equivalent perturbations, ensuring that the functional correctness of the code remains unchanged.

The primary reason for this strict requirement is that unit tests are commonly used to verify correctness when assessing the robustness of code generation models [Wan+23; Mas+23; Yan+23a]. If a perturbation were to alter the semantics of the prompt, it would naturally lead to semantically different outputs, making unit test comparisons invalid.

Despite this framework not relying on unit test correctness, the requirement remains important. Evaluating an LLM with semantically different perturbations would not measure robustness, but instead test the model's capability to translate a different intent. This does not align with the robustness definition of the thesis.

Real-World Relevance

Another requirement for the perturbation strategies in this thesis is real-world relevance. To confidently evaluate a code translation's robustness for practical usage, the perturbations have to account for the variances that are relevant in well-intentioned real-world scenarios. The thesis's robustness evaluation assumes that a well-intended code translation environment does not need to prove robustness against adversarial attacks, as would be common in adversarial robustness.

Instead, a robust model should deviate in performance when translating a variety of *C* projects into *Rust*, even when faced with projects of teams with diverse coding styles and conventions. This may also include teams with varying programming experience, which results in different levels of code clarity, as well as real-world noise by inconsistencies in formatting, naming conventions, and structure.

To reflect these challenges, the chosen perturbation strategies must capture realistic variations in code. This includes not only superficial transformations (e.g., changes to identifiers in level II of Figure 3.1) but also deeper structural modifications (e.g., changes in control flow in level VI of Figure 3.1) while preserving semantic equivalence.

Syntactical Correctness

When automatically modifying existing code, perturbation strategies can easily introduce syntax errors. However, the framework's design strictly requires maintaining syntactical correctness for the perturbed *C* code.

Evaluating the *functional equivalence* between the *C* and *Rust* requires that both code versions are compilable. If the perturbed *C* code cannot be executed, it becomes impossible to compare its behavior against the translated *Rust* version. This makes syntactical correctness a crucial requirement when wanting to evaluate the robustness of *functional equivalence* between translation and original code.

Trade-offs in Perturbation Selection

While various perturbation strategies exist in related works, implementing all of them is impractical. Since none were originally designed for *C*, adapting them requires a full re-implementation, which exceeds the scope of this work.

Additionally, applying SOTA robustness metrics such as RP_s@k has its drawbacks. Each perturbation requires executing translations $F \times s \times n$[1] times. This results in high computational and financial costs, depending on whether paid APIs or local models are used.

Therefore, selecting meaningful perturbations is essential to balance feasibility and coverage. The chosen perturbation strategies have to meet the requirements while representing a broad range of transformation magnitudes to accurately reflect possible code variations of the real world.

4.2.2 Categorizing Perturbation Strategies

To systematically assess robustness, perturbation strategies are categorized in three different properties: (*i*) *perturbation determinism*, (*ii*) *perturbation target*, and (*iii*) *perturbation level*.

Perturbation Determinism

Some perturbation strategies inherit randomized operations, whereas others act in a static way. Consequently, this work distinguishes between *stochastic strategies* and *deterministic*

[1]Where F is the number of files in the original benchmark dataset, and s, as well as n, follow the definition in RP@k.

strategies. This categorization is important when measuring the robustness, because evaluating *stochastic strategies* introduces a problem. Imagine having a perturbation strategy **Butterfinger** that randomly produces typos to an original instruction as illustrated in the following prompts.

Original Instruction

Translate the following C code to Rust.

Butterfinger Instruction Version 1

Translate fhe foolowing C code to Rust.

Butterfinger Instruction Version 2

Translate the following C code to Tyst.

The general task of *Version 1* remains recognizable for a human and probably also for an LLM. However, while *Version 2* also solely introduces two character replacements, the objective of the task becomes hard to understand. Since *Tyst* is no real programming language, the model could be unsure whether the user originally meant *Typescript* or *Rust*, which could produce results not aligned to the actual goal of the task. Having the chance of producing easier or more difficult versions of a perturbation poses the question of which one of the versions to choose for the evaluation.

The naive way would be to choose the hardest prompt, but this is not directly measurable beforehand. Hence, *ReCode* proposed metrics relying on the worst-case approach, such as RP_s@k, where a task is only considered correct if and only if it produces a correct result, for s different versions of the perturbation strategy.

This presents a good approach to incorporate the variances of *stochastic strategies* into the evaluation. However, applying this approach to *deterministic strategies* is not necessary, since there are no differing variants. This would unnecessarily increase the amount of LLM calls, resulting in higher expenditure in cost and computation, an aspect that will be discussed in Section 4.3.4. Consequently, a distinction between these strategies is beneficial when evaluating the robustness. In consequence, the approach of using s versions for a *stochastic perturbation* affects the whole process of Step I, which is particularly described in Section 4.2.4.

Perturbation Target

Besides being *deterministic* or *stochastic*, perturbations are also categorized based on the target they aim to create change:

- Instruction: Perturbations applied to the natural language instruction that describes the task to the model (see Section 2.4.1).
- Comments: Perturbations applied to comments inside the code sample.
- Code: Perturbations applied to transform the code sample.

As stated in Section 3.3, evaluating the robustness to the natural language instruction part is not that meaningful in the context of LLM-based code translation. A deployed code translation system would only ask for the source and target languages in addition to the code that is to be translated. Hence, when creating such a system, one task would be to identify the optimal task description, which stays static afterwards. Neither the creation of such a system nor the optimization process is part of this robustness evaluation.

Consequently, this work focuses more on identifying perturbations on code and comments. However, excluding instruction perturbations completely would mean leaving out the opportunity to compare whether code and comment perturbations will have a similar impact as instruction perturbations. Specifically, an instruction to perform LLM-based code translation could be defined as: “Translate this *C*-Code into *Rust*”. In contrast to the whole prompt, which contains this instruction and the code part, this is only a fraction of the prompt. However, since each word of this sentence has quite a lot of impact on the semantics of the instruction, perturbing this slightly could result in drastically different outputs. This hypothesis has yet to be shown empirically for code translation.

Superficial vs Profound - Levels of Perturbations

Apart from perturbation target and perturbation determinism, perturbations targeting comments or code are further categorized into the six levels by Faidhi and Robinson [FR87], as described in Section 3.3.3. Using the six levels is not only motivated by the fact that it helps to distinguish strategies impact-wise[2]. Instead, it also helps to identify strategies in general. Categorizing into the six levels shows immediately whether there are strategies that change not only simple properties like identifiers, but also perform deeper structural transformations. When evaluating the robustness of code perturbations, this is an important aspect.

According to the definition of robustness, a robust code translation model should be capable of producing similar results, no matter if the code is written in a different structure and different decision logic. Since different programmers tend to create different solutions to the same problem, which has been empirically shown by Nanz and Furia [NF15][3], it is important part of the evaluation to test results with structurally different code inputs. Using such level-based categorization adheres to the trade-offs in perturbation selection, as it shows which kind of perturbations are necessary to represent various relevant scenarios, without involving too many perturbations that perform similar things.

4.2.3 Implemented Perturbation Strategies

This subsection lists the implemented perturbations and briefly describes the motivation behind choosing these strategies. Each perturbation strategy is described in full detail in Appendix A.1, where the explanation highlights that each strategy adheres to the requirements of a perturbation strategy. Table 4.1 gives an overview of the strategies. Since some strategies can be found in previous work, the table contains a related work column that references papers that used a perturbation strategy similar to the one. Table 4.1 demonstrates, that some perturbation strategies are performed on either *instructions*, *comments* or *code*. Although the framework technically allows a single perturbation to

[2]In this case, the impact is referred to as a prompt being either superficial or profound, with more changes on structure or control flow.

[3]Altough this study only evaluated different solutions to the same task in different programming languages, this leads to the idea, that solutions can be different, based on which language a programmer mostly works with.

Strategy	Determinism	Target	Level	Related Work
Backtranslation	Deterministic	Instruction	-	Mastropaolo et al.[Mas+23] Huang et al.[Hua+21]
Butterfinger	Stochastic	Instruction	-	
ChangeCharCase	Stochastic	Instruction	-	
Concretizer	Stochastic	Instruction	-	COCO[Yan+23a]
Translation	Deterministic	Instruction	-	
Backtranslation	Deterministic	Comments	I	ReCode[Wan+23]
Butterfinger	Stochastic	Comments	I	ReCode[Wan+23]
ChangeCharCase	Stochastic	Comments	I	ReCode[Wan+23]
LLMCommentInsertion	Stochastic	Comments	I	
RemoveComments	Deterministic	Comments	I	Zhang et al.[Zha+23b]
Translation	Deterministic	Comments	I	
CodeFormat	Deterministic	Code	I	
ABC	Deterministic	Code	II	Yu et al.[YWW22]
Backtranslation	Deterministic	Code	II	
Butterfinger	Stochastic	Code	II	ReCode[Wan+23]
CamelCase	Deterministic	Code	II	ReCode[Wan+23]
ChangeCharCase	Stochastic	Code	II	ReCode[Wan+23]
LLMVariableImprove	Stochastic	Code	II	
IdenObfuscator	Deterministic	Code	II	
PascalCase	Deterministic	Code	II	
SnakeCase	Deterministic	Code	II	
Translation	Deterministic	Code	II	
ConstantInsertion	Stochastic	Code	III	SPPlagiarise[CLS19]
DeadCodeInsertion	Stochastic	Code	III	ReCode[Wan+23] Zhang et al.[Zha+23b]
IncludeCommentAdder	Stochastic	Code	III	
LLMCodeExtraction	Stochastic	Code	IV	SPPlagiarise[CLS19]
FunctionSignatureChange	Stochastic	Code	IV	
ForWhileSwitch	Deterministic	Code	V	ReCode[Wan+23], SPPlagiarse[CLS19], Li et al.[Li+24b] Zhang et al.[Zha+23b] Yu et al.[YWW22]
ConditionSwap	Deterministic	Code	V	ReCode[Wan+23] Li et al.[Li+24b] Zhang et al.[Zha+23b] Yu et al.[YWW22]
ConditionDup	Stochastic	Code	VI	Li et al.[Li+24b]
DeMorgan	Deterministic	Code	VI	

Table 4.1: Overview of implemented perturbation strategies, categorized regarding determinism, target, and level. Perturbation strategies that were motivated by previous works reference their originating related work.

target multiple places in parallel, the decision was to always select only one target. This improves the interpretability of the evaluation results.

The Appendix additionally contains a Table that demonstrates the perturbations on an exemplary prompt.

As stated previously, the perturbation strategies have to be real-world-relevant and reflect input variations that could appear in practice. The perturbations **ChangeCharCase** and **Butterfinger** represent common errors that appear and present regular noise. With **CodeFormat**, **CamelCase**, **PascalCase**, and **SnakeCase**, the influence of different coding conventions and style guides on code translation performance is examined. Furthermore, the effect of different levels of documentation is assessed by using **RemoveComments** and **LLMCommentInsertion**. By applying **Concretizer**, **Backtranslation**, **LLMVariableImprove**, **LLMCodeExtraction**, or **ForWhileSwitch**, the robustness to individuality in code and prompts written by different persons is evaluated. **ConditionSwap** and **DeMorgan** go even further by assessing the same in decision logic. Furthermore, with **Translation**, the framework examines robustness for multilingual scenarios. Lastly, **IncludeCommentAdder**, **ABC**, **IdenObfuscator**, **ConstantInsertion**, **DeadCodeInsertion**, **FunctionSignatureChange**, or **ConditionDup** represent perturbations that simulate a poorly written or poorly documented codebase, which is part of the real world.

With these perturbation strategies, the framework covers a broad spectrum of input variances that follow different objectives and also represent different targets and levels of such variances.

Table 4.1 highlights that most of the perturbation strategies are applied on the code part of the prompt, i.e., comments or code, which addresses the identified research gap, missing a comprehensive evaluation of code perturbations with different complexities. Considering this, the table also shows that most of the perturbation strategies act on superficial levels I and II. However, compared to previous robustness evaluation works[4], this framework incorporates a higher number of profound strategies. Only *ReCode* evaluated the robustness on perturbations that were deeper than level II, i.e, three strategies, whereas this framework incorporates nine perturbations that are deeper than level II. Despite having more profound perturbation strategies, the levels are still not evenly distributed.

This is because, when wanting to automatically generate semantically equivalent perturbations, the difficulty and implementation complexity increases with each level. Having at least two perturbations for each level seemed to be a good trade-off between covering a broad range of variations and not creating overly complex strategies, which aligns to Section 4.2.1.

4.2.4 The Perturbation Process

Implementing and choosing the right perturbations is one part of Step I. Another one is utilizing the implemented perturbation strategies to produce perturbed datasets that can be used for Step II and Step III.

For this, Step I expects a dataset and a configuration as input. The dataset contains multiple *C* files, with which the code translation robustness should be assessed. Such a dataset must be carefully designed to comprehensively assess robustness in code translation and account for the research gap “programming languages and benchmark complexity”.

[4]Referring to the works that were described in Chapter 3 (*ReCode* [Wan+23], Mastropaolo et al. [Mas+23], *COCO* [Yan+23a], and Improta et al. [Imp+25]).

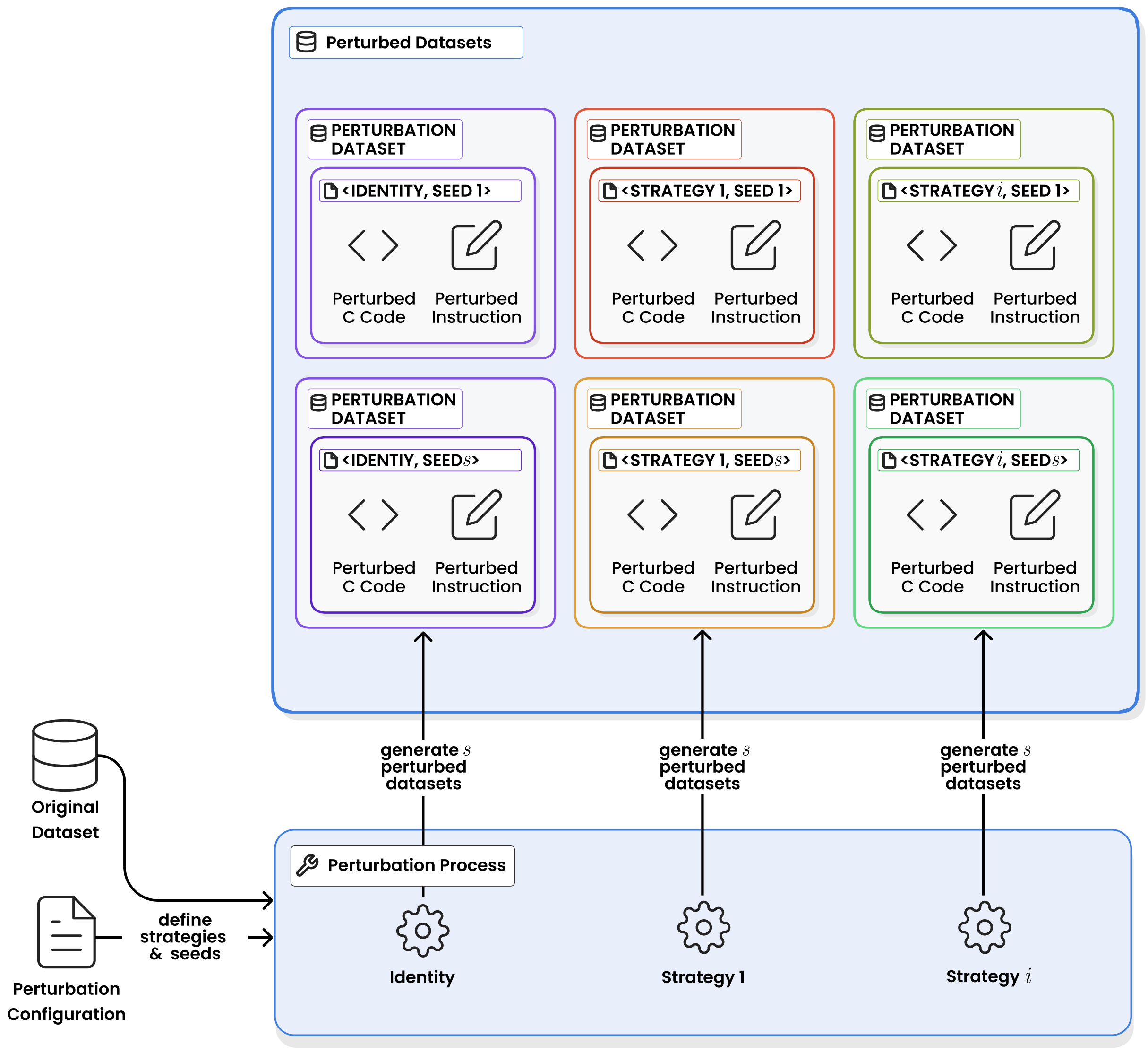


Figure 4.2: Perturbation process of the framework, using s seeds and i perturbation strategies, creating various perturbation datasets.

The files should cover a diverse range of complexities and real-world relevant *C* source code to ensure that the robustness of the system is accurately measured for practical usage.

The additionally entered configuration file specifies the perturbation strategies and their parameter settings. Furthermore, it contains the original natural language instruction and a seed value, to make results reproducible and also allow the creation of differing *stochastic perturbation strategies*.

Specifically, comprehensively evaluating *stochastic strategies* requires assessing a model's performance on multiple variances of stochastically perturbed inputs, as presented in Section 4.2.2. This is achievable by using multiple seed values for the perturbation process when doing the robustness assessment. Subsequently, the number of different seeds is denoted as s, similar to *ReCode*'s notation [Wan+23].

Figure 4.2 illustrates the process of creating the perturbed datasets. The visualization shows that for each seed, there is one additional dataset: **Identity**. This describes the original prompt, without perturbation. While it is technically not necessary to create such a **Identity** dataset or even multiple identical **Identity** datasets, it is used to reduce implementation effort. The dataset evaluated in this thesis is not huge, hence doing it like this is an okay trade-off and follows the *YAGNI*[Fow15] principle. When wanting to evaluate very large datasets, one should consider changing this and working with the files directly out of the original dataset.

For the evaluation of *deterministic perturbations*, s should be set to 1. Otherwise, one would produce multiple identical datasets for the same perturbation strategy. Evaluating these identical datasets with a worst-case metric, such as $\mathrm{RP}_s@\mathrm{k}$, would not follow the concept that these s datasets describe different datasets.

Hence, when designing the experiments, they either evaluate *deterministic perturbations* or *stochastic perturbations*, because one of them involves multiple s datasets and the other does not.

For i perturbation strategies[5], and s seeds, the implemented framework will consequently create $s \times i$ datasets, which ultimately results in $s \times i \times F$ prompts. Consequently, there is a perturbed dataset for each pair of $\langle strategy, seed \rangle$[6].

Specifically, a perturbed dataset includes not only the perturbed *C* file, but also the belonging instruction. Since some perturbation strategies like **Concretizer** require an instruction tailored to a specific file, for each *C* file, there is an accompanying instruction file. This instruction file has the same basename as its *C* file but ends in `.inst`, which ensures a one-to-one pairing.

Appendix A.1.1 goes into more details of implementing perturbation strategies and illustrates that such automatic perturbations are not trivially implementable. Additionally, it details that the perturbation process incorporates an automatic syntax check, which directly notifies the user in case a perturbation does not align with the syntax-correctness requirement.

4.3 Step II - Translation

In Step II, all perturbed *C* code prompts generated in Step I are passed to an LLM-based code translation system that translates them into *Rust*. Specifically, this system is a SOTA LLM-based *C-to-Rust* translation system that has been designed by *Bosch* [QHW25]. This system not only produces the actual *Rust* translations for every perturbed prompt, but also the translation statistics (e.g., compilation or fuzzing success). These outputs are then used in Step III to measure the model's robustness and consequently prepare the framework to be used in experiments to answer the RQs.

4.3.1 Overview and Goals

Recall that Step I and its perturbation strategies generate many perturbed versions of a single original file to enable a comparison of the translation results on the original and perturbed prompts. Step II aims to process the results of Step I and produce the LLM-based *Rust* translations to prepare a comprehensive evaluation in Step III.

The high-level overview of Step II can be described as follows.

At first, iterate over each perturbed file and its instructions. Second, call the LLM to translate the file from *C* to *Rust*. Lastly, collect the LLM's output and store both

[5]Where **Identity** is included in i.

[6]Where strategy refers to perturbation strategy.

the produced *Rust* code and any relevant statistics. Specifically, those statistics involve whether the *Rust* code compiles, and if the translation passes the *differential fuzzing*.

Since LLMs are nondeterministic and the same prompt may lead to different results, each prompt is translated n times. This allows the framework to apply metrics like *pass@k*.

Additionally, Step II applies a *feedback loop* strategy. Whenever the generated *Rust* code fails to compile or does not pass the *differential fuzzing* check, the system re-prompts the model with the encountered error. However, this process is limited to a predefined number of iterations to prevent infinite loops.

This mimics how a programmer might inform the model about an error and request a correction, consequently improving the chance of better results.

The following subsections detail how prompts are structured as input, how outputs are saved, and how the translation process (including verification and the *feedback loop*) is implemented.

4.3.2 Input Specification

The input of Step II consists of the various perturbation datasets produced in Step I. Specifically, such a dataset contains the perturbed and syntactically correct *C* files and the accompanying instruction files.

Step I not only produces one perturbation dataset for each strategy, instead it may also produce s different versions of each strategy when working with *stochastic perturbations*.

The framework organizes them into distinct datasets, one for each $\langle strategy, seed \rangle$ pair. Each of these datasets contains the *C* code and instruction files to be used for the code translation prompt.

As Figure 4.1 highlights, Step II not only receives the results of Step I, but also involves a configuration. This configuration specifies the LLM to use for the translation, which is an important feature to examine RQ5. Furthermore it includes the parameters: *max_retries*, *fuzzing_time*, and n, that are detailed in Section 4.3.4

4.3.3 Output Specification

For each dataset, Step II must produce both translated *Rust* files and translation statistics.

Translated Rust Files For every pair of $\langle strategy, seed \rangle$ Step II saves each of the *Rust* translations to disk.

Translation Statistics For every $\langle strategy, seed, run \rangle$ pair, where *run* denotes a single attempt in n for *pass@k*, Step II saves translation statistics in a `.csv` file [Sha05], that contains statistics for each $\langle prompt, run, iteration \rangle$ pair that is processed. The *iteration* refers to the number of retries in the *feedback loop*, that is detailed in the next section.

These statistics include whether the LLM produced *Rust* code at all, whether the *Rust* code compiled successfully (*compilation success*), and whether the differential fuzzing did not result in any semantic mismatches (*fuzzing success*). Additionally, the number of tokens and possible exceptions are logged.

The statistics are the foundation for the final robustness metrics in Step III, which build upon *pass@k* and RP_s@k.

Algorithm 1 The code translation process of Step II that utilizes the perturbed datasets and generates Rust files, as well as translation statistics.

```
 1: $max\_retries \leftarrow 5$
 2: for all perturbation strategy $i$ in perturbation datasets do
 3:     for all perturbation version $s$ in strategy $i$ do
 4:         for $run \leftarrow 1$ to $n$ do
 5:             initialize translation statistics for $(i, s, run)$
 6:             for all $prompt_F$ in perturbation dataset $(i, s)$ do
 7:                 $translation \leftarrow \mathrm{LLM}(prompt_F)$
 8:                 $iteration \leftarrow 0$
 9:                 $SaveRust(translation)$
10:                 $RecordStats(F, run, iteration, translation)$
11:                 while $translation$ invalid and $iteration < max\_retries$ do
12:                     $prompt'_F \leftarrow prompt_F + translation.error\_msg()$
13:                     $translation \leftarrow \mathrm{LLM}(prompt'_F)$
14:                     $SaveRust(translation)$
15:                     $iteration \leftarrow iteration + 1$
16:                     $RecordStats(F, run, iteration, translation)$
17:                 end while
18:             end for
19:         end for
20:     end for
21: end for
```

4.3.4 The Translation Process

Algorithm 1 outlines the translation process in Step II.

At first, the process iterates over all perturbations. Meaning for each perturbation strategy i and each of its s versions, the framework collects the code translation prompts from Step I.

Each of these F prompts is then translated n times, to gather multiple translation results, capturing the nondeterminism of the LLM. The value of n is specified via the translation configuration (see Figure 4.3).

Lastly, each LLM response is verified for *compilation success* and *fuzzing success*. If either check fails, the process goes into the *feedback loop*. This means it uses the error message of the verification to redefine the prompt and guide the model towards a valid translation. This *feedback loop* process gets repeated until the translation is valid, or the number of iterations exceeds the specified maximum amount.

In detail, the maximum number of LLM calls in Step II can be calculated with:

$$max\_llm\_calls := i \times s \times n \times F \times max\_retries. \tag{4.1}$$

Consequentially Step II produces $i \times s \times n$ translation statistic `.csv` [Sha05] files.

Figure 4.3 illustrates the key steps involved when processing a single code translation prompt, which happens for each $\langle strategy, seed, run \rangle$ pair. The following sections describe the key steps of *verification* and the *feedback loop* in more detail.

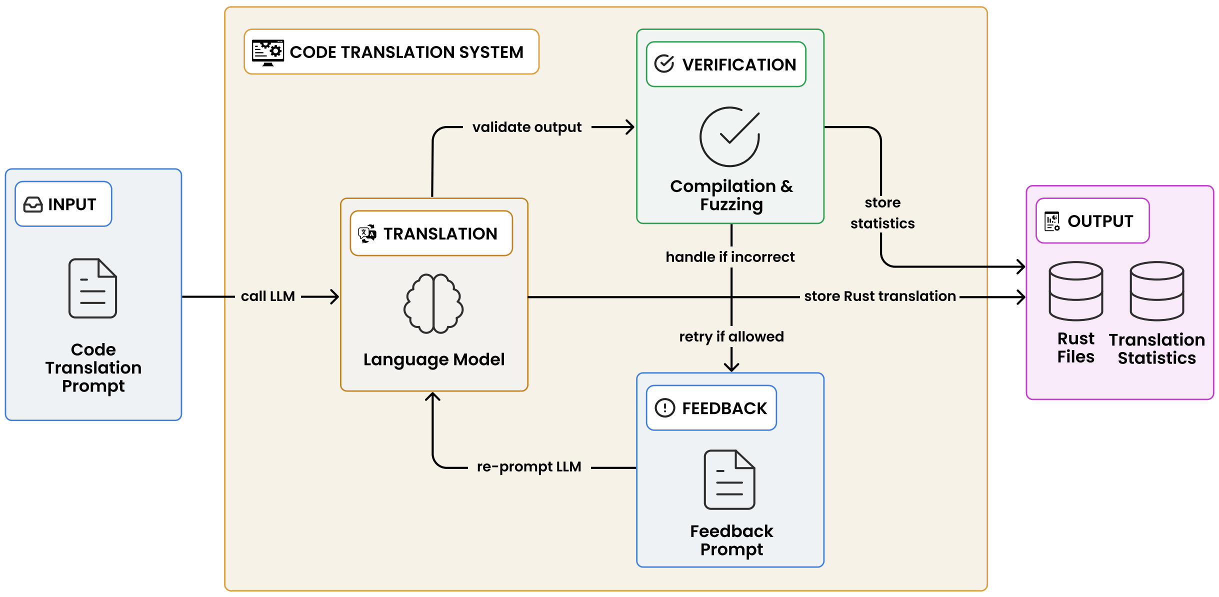


Figure 4.3: Process of a single translation with the code translation system of the framework.

Verification and Feedback Loops

As detailed in Figure 4.3, after receiving an LLM response, the pipeline performs a verification, which involves the criteria *compilation success* and *fuzzing success*. In case the verification fails, the original prompt is redefined and extended by an error message. The experiments in this thesis will be conducted with a *max_retries* of five. Earlier testing showed that initial errors are mostly resolved in the first few iterations. If an error persists beyond this point, the model tends to reproduce it consistently, indicating an inherent limitation in its ability to correct the issue. While this theoretically could miss on a few later improvements, five seemed to be a good trade-off between maximizing correction attempts and avoiding unnecessary costs, as each additional iteration directly impacts the number of LLM calls in Equation (4.1). Chapter 7 will examine whether *max_retries* of five empirically presents a good trade-off for robustness evaluation experiments.

As described before, the statistics of each iteration of the loop will also be added to the output `csv`. This enables an evaluation of the system with and without the *feedback loop*. Consequently, the framework can evaluate whether using such a strategy has an impact on the robustness of LLM-based code translation, which will be answered in Chapter 7 and accounts for RQ3.

Specifically, the two correctness criteria are verified as follows.

Compilation Success The generated *Rust* code must compile. If it fails, the compiler error message is appended to the prompt, asking the model to fix it. Specifically, the compilation is verified by using *rustc* [Thec], which produces a list of errors in case the *Rust* code is not compilable.

Besides verifying the compilation with *rustc*, the code is examined regarding linting errors with *clippy* [Lan]. Similar to *rustc*, *clippy* also returns a list of error messages, in case the generated code is not free of linting errors.

If errors are found, the prompt is extended as follows:

Prompt with Compilation Error

You made the following mistakes:

```
<rustc|clippy error message 0>
<rustc|clippy error message 1>
<rustc|clippy error message ...>
```

Fuzzing Success The pipeline uses *differential fuzzing* to compare the behavior of the prompted *C* code against the generated *Rust* code, thereby testing their *functional equivalence*. The implemented *differential fuzzing* approach is similar to the one used in *FLUORINE* [Eni+24], visualized in Figure 2.3. However, instead of generating fuzzing inputs for *Rust*, the used system generates fuzzing inputs for *C*.

If the fuzzer finds a *counterexample* that demonstrates a behavioral mismatch and that specific example is appended to the prompt of the next iteration.

Specifically, the *differential fuzzer* tries finding *counterexamples* for all identically named functions in the *C* and *Rust* code. This process is done for a specified number of seconds for each function in a file. The amount is configured by the *fuzzing_time* parameter in the configuration of Step II. In the experiments of this thesis, the *fuzzing_time* is strictly set to 15 seconds. As the code translation system is a predefined system, it has already undergone some testing. Specifically, different *fuzzing_time* values were tested, but it turned out that most *counterexamples* were identified rather quickly. According to prior testing, fuzzing times beyond 15 seconds mostly led to additional inputs being fuzzed without identifying further *counterexamples*. Since fuzzing is applied to all results of each LLM response, i.e., Equation (4.1), choosing an unnecessarily long *fuzzing_time* would drastically increase the runtime of the entire pipeline. Therefore, 15 seconds presented an effective *fuzzing_time* setting.

In case the fuzzer identifies one or more *counterexamples*, the feedback prompt is structured accordingly:

Prompt with Fuzzing Counterexamples

You made the following mistakes:

```
In function <function name>:
    Fails for input <parameter_name> = <counterexample_value>
```

Note that all identically named functions are being fuzzed, and the fuzzer can identify *counterexamples* for more than one function.

Furthermore, the implementation of a fuzzer that heuristically compares various inputs for certain parameters is not trivial, and it may lead to problems in certain edge cases and throw exceptions. In such cases, the LLM response is taken as is, and there is no further *feedback loop* iteration. When evaluating the experiments, it will be discussed whether fuzzing failures resulted because of *counterexamples*, or *fuzzing exceptions*.

4.4 Step III - Evaluation

The third step focuses on evaluating the produced translation statistics of Step II, therefore quantifying the system's robustness. Specifically, this step provides the necessary tools and metrics to comprehensively evaluate the robustness of an LLM-based code translation system.

4.4.1 Overview and Components

As Figure 4.1 shows, Step III gets both the perturbed prompts from Step I and the translation statistics and generated *Rust* files as input. In contrast to the previous steps, Step III does not provide a fixed processing pipeline, but rather a collection of evaluation components that can be flexibly combined to analyze different nuances of robustness that are existing research gaps (see Section 1.2.5).

These components are: (*i*) robustness metrics based on *pass@k* and its extensions, (*ii*) aggregation methods for translation statistics, and (*iii*) similarity analysis between original and perturbed prompts.

With these components, the framework provides all the necessary tools to assess the robustness of the used LLM in Step II. While there is no specific component that directly distinguishes between nondeterministic model noise and genuine robustness deficits, the framework can be used appropriately to enable such differentiation. Section 4.5 details how the framework can and should be used for a comprehensive robustness evaluation.

4.4.2 Robustness Metrics

The core component of Step III is the implementation of metrics for quantifying the robustness.

Pass@K and Its Extensions

The underlying metric used in this step is *pass@k*, which has been introduced in [Che+21]. This metric indicates the probability that at least one of k samples provides a correct result in n runs, with $n \geq k$. For the evaluation of robustness, *ReCode* extended the metric to new variants such as RP_s@k.

The thesis utilizes the concepts of RP_s@k and defines a new extension: *Robust Change$_s$@k* (RC_s@k). The next part describes the meaning of the metrics used in the context of the thesis.

Robust Pass@k Measures the probability that a code translation system provides a successful translation for all s versions of a *stochastic perturbation*. This is specifically relevant for *stochastic perturbations*, which has been reasoned before. Since *deterministic perturbations* do not provide variants, these perturbations are evaluated with $s = 1$. By definition, $RP_s@k$ with $s = 1$ is equivalent to $pass@k$, and for uniformity and better understandability, the thesis always uses the term $RP_1@k$ instead of $pass@k$, also for the *deterministic perturbations*.

Robust Change@k Quantifies the relative change in translation performance between the original **Identity** and a perturbation. The metric is defined by Equation (4.2).

$$RC_s@k = \frac{|RP_s@k_{Identity} - RP_s@k_{Perturbation}|}{RP_s@k_{Identity}} \tag{4.2}$$

This metric directly reflects the amount of performance deviations observed for a perturbation. Since both performance increase and decrease demonstrate robustness deficiencies, the decision was to use the absolute value. While it is neat to directly observe whether a perturbation improved or decreased performance, it comes with drawbacks when wanting to aggregate or intuitively visualize information.

Since the comprehensive evaluation uses both metrics, by comparing $RP_s@k$, it can be directly observed whether a perturbation produces better or worse results.

Note that these metrics can be applied to both *compilation success* and *fuzzing success*. However, *fuzzing success* is the more relevant success metric as it strictly defines that a translation is functionally equivalent. For the real-world application of a code translation system, this is the final result that is desired. Nonetheless, *compilation success* can be used to improve the interpretation, why certain tasks or perturbations might fail.

The selection of these two metrics, $RP_s@k$ and the newly defined $RC_s@k$, was made since they are closely aligned with the goals of this thesis. $RP_s@k$, based on the approach of *ReCode* [Wan+23], serves as an established metric to quantify the absolute robust performance considering stochastic variations ($s > 1$) and as an equivalent to *pass@k* for *deterministic* perturbations with $s = 1$. In order to directly measure the model's sensitivity to specific perturbations, $RC_s@k$ was introduced. The metric measures the relative change in performance compared to the baseline $RP_s@k_{Identity}$.

These two metrics are sufficient to comprehensively answer the RQs of this thesis, without overloading the analysis with a large number of metrics. The combination allows an assessment of both the absolute robust performance and the specific effects of individual perturbations.

4.4.3 Aggregation Methods

With the different perturbation characteristics, the framework enables various options for aggregating the metrics.

Recall that a perturbed dataset consists of F files. The default metric $RP_s@k$ is computed for each file x in the dataset and then aggregated by taking the mean:

$$RP_s@k = \frac{1}{F}\sum_{x=1}^{F} RP_s@k(x).$$

This is the default aggregation method, used when reporting the performance for an entire perturbation or **Identity**.

Since the perturbations have different properties (such as *determinism*, *target*, or *level*), the aggregation can be further specialized. For a given property, let P denote the number of perturbations that share the same characteristic. Then, the aggregated metric for that property is defined as:

$$RP_s@k_{property} = \frac{1}{P}\sum_{p=1}^{P} RP_s@k(p),$$

```
int sum(int num1, int num2) {
    return num1 + num2;
}
```

Listing 4.1: Identity sum function.]Simple **Identity** sum function.

```
int sum(int number1, int number2) {
    return number1 + number2;
}
```

Listing 4.2: Simple *perturbed* sum function.

where $RP_s@k(p)$ is the aggregated dataset performance for perturbation p among the P perturbations with that property.

These aggregations enable a systematic evaluation and can be leveraged to compare robustness impacts caused by perturbations of different characteristics. For example, it may show that deeper structural changes to code may cause more robustness issues than the superficial levels I or II.

4.4.4 Semantic Similarity Analysis

Section 3.4.6 detailed that related work always incorporated a way of measuring similarity between original and perturbed prompt versions. However, none of the works examined whether stronger perturbations on code also lead to stronger robustness deviations. To close this gap Step III includes a component to measure the semantic similarity between the original and perturbed prompts.

Motivation

Section 4.2.1 shows that a key requirement in a perturbation-based robustness evaluation is that each perturbation keeps its semantical meaning. If not, the framework would not be testing the model's robustness to similar inputs, but rather its ability to handle an entirely different problem, which is not relevant in terms of robustness.

However, even when two files are semantically equivalent, there can still be major differences in how the code is structured. Consider the following example of an unperturbed, simple addition function in Listing 4.1 and a highly similar perturbed version in Listing 4.2, as well as a complex perturbation in Listing 4.3. While all inherit the same functionality[7], Listing 4.3 is likely perceived as far more complicated. If a model cannot translate the bitwise version with similar performance to the simple version, labeling this issue equally non-robust as if the model would fail on the easier perturbation may be misleading.

In real-world usage, such details matter because users expect the code translation system to be robust to small changes in syntax, layout, or style. Minor edits like renaming variables should not lead to large differences in the translated output. However, if a perturbation drastically rewrites a function, the model might show a drop in performance that does not necessarily indicate a lack of robustness, but rather reflects the higher difficulty of translating significantly more complex code. This leads to RQ4, which investigates how

[7]Specifically, the bitwise variant may work differently on non-32-bit hardware.

```
int sum(int num1, int num2) {

    // 32 bit mask in hexadecimal
    long mask = 0xffffffff;

    // Iterate till there is no carry
    while ((num2 & mask) != 0) {

        // carry contains common set bits of num1 and num2, left
           shifted by 1
        int carry = ((num1 & num2) & mask) << 1;

        // Update num1 with (num1 + num2 without carry)
        num1 = num1 ^ num2;

        // Update num2 with carry
        num2 = carry;
    }
    return num1 & mask;
}
```

Listing 4.3: Complex semantically equivalent sum perturbation [Gee24].

semantic similarity between original and perturbed inputs relates to observed robustness deficits.

Similarity Measure

There are multiple ways to measure the similarity between text and code. This itself is an entire research area that could be explored and compared. However, measuring the similarity between prompts is only a nuance of a comprehensive robustness framework and is therefore not explored in full detail. Specifically, the thesis chose among similarity approaches that have been incorporated in robustness-related work, namely *Levenshtein distance* [Lev66], *CodeBLEU* [Lu+21; Ren+20], and *cosine similarity* [SWY75b] on embedding vectors [Mik+13a].

Among these techniques, *cosine similarity* on embedding vectors reflects the desired similarity the best. As detailed in Section 2.2.1, embedding models try to capture the semantics of words or sentences to encode them into a vector representation. Consequently, vectors with a greater distance from each other suggest less similar inputs than vectors closer to each other. While *Levenshtein distance* and *CodeBLEU* also measure how much the text has been changed, they tend to focus on literal token differences or syntactic differences. By contrast, embeddings and *cosine similarity* allow for directly reflecting a model's internal representation of code. The hypothesis is that this can highlight deeper semantic or stylistic similarities that token-based methods may miss.

To better understand why *cosine similarity* is the best fit for quantifying similarity in a perturbation-based robustness evaluation, the next examples detail the superiority of *cosine similarity* compared to the other approaches.

Recall the **Butterfinger** instruction example of Section 4.2.2. By calculating the similarity with *Levenshtein*, the measure would suggest that both perturbations are equally similar to the **Identity**, with a literal difference of two (Table 4.2). However, this does not accurately reflect the semantics of the instruction, as desired. The intent of the task is

Perturbation	**Levenshtein ↓**	**BLEU ↑**	**Cosine Similarity ↑**
Translate the following C code to Rust.			
Translate fhe foolowing C code to Rust.	2	0.541	0.953
Translate the following C code to Tyst.	2	0.707	0.826

Table 4.2: Comparing *Levenshtein* [Lev66], *BLEU* [Pap+02; LO04] and *Cosine Similarity* for a **Butterfinger** perturbation example on "Translate the following C code to Rust.". The used embedding model is *OpenAI*'s *ada-002* [Ope22c].

more difficult to understand when the target language *Rust* is not clearly defined.

Furthermore, *BLEU*[8] [Pap+02; LO04] captures that both perturbations are different, as their value range is from zero to one, where one represents identical inputs. However, it also states that the seemingly easier perturbation for the model is less similar.

In contrast, *cosine similarity* accurately reflects the similarity differences that would be desired. Considering that the *cosine similarity* yields values from minus one to one, it shows that the first example is close to the original instruction, and the one with the modified target language is significantly less similar.

Perturbation	**Levenshtein ↓**	**CodeBLEU ↑**	**Cosine Similarity ↑**
Listing 4.2	12	0.534	0.968
Listing 4.3	430	0.574	0.908

Table 4.3: Comparing *Levenshtein* [Lev66], *CodeBLEU* [Lu+21; Ren+20] and *Cosine Similarity* for perturbations on code for the example of Listing 4.1. The used embedding model is *OpenAI*'s *ada-002* [Ope22c].

The same example can be shown for code, by comparing the samples for the sum function of Listing 4.1 in Table 4.3. While *Levenshtein* accurately reflects that Listing 4.3 is less similar, this is only because the number of characters is higher. A perturbation that adds very elaborate comments could even yield higher *Levenshtein* scores, which does not reflect the similarity that is desired for a robustness evaluation. In addition, *CodeBLEU* measures that both perturbations are quite different from the **Identity** and even indicates that the code in Listing 4.2 is less similar than that in Listing 4.3. The *cosine similarity* value shows the expected comparison, where Listing 4.3 yields lower similarity than Listing 4.2. Consequently, *cosine similarity* seems to reflect the similarity that is desired for the robustness evaluation.

This ultimately leads to RQ4: "What is the correlation between semantic similarity and perturbation-based robustness?", which aims to clarify whether bigger differences in these embedding-based similarity scores align with larger robustness deficits, or whether the model's performance is unpredictable even when the embeddings show high similarity. If there is a clear correlation, the *cosine similarity* could be utilized a priori to exclude perturbations that produce modifications that mislead a robustness evaluation.

[8]BLEU is the original metric that has later been extended to *CodeBLEU* to measure the similarity of code.

Similarity Baseline

While Table 4.2 and Table 4.3 showed that *cosine similarity* accurately reflects the desired similarity differences, the actual values are hard to interpret by themselves. The *cosine similarity* can, in theory, produce values ranging from minus one to one, yet the examples only showed values ranging from 0.826 to 0.968.

To get a better understanding of the values an embedding model produces, Step III involves a component that creates a baseline. This baseline gives information about the relevant value range of *cosine similarities* for a given model. This value range distribution can be derived by performing pairwise comparisons of all C files (F) in the benchmark dataset. Since the F files are different from each other, the computed similarity scores reflect how the embedding model interprets semantically different files that share only the general property of being C code.

This value distribution can be utilized for statistical operations. Since the distribution shows values for semantically different files, high *cosine similarities* that represent statistical outliers to this distribution should therefore describe files that are more similar than the other pairwise comparisons.

A common practice to detect such outliers is by utilizing the *Z-Score* [08].

$$\text{Z-Score} = \frac{x_i - \mu}{\sigma} \tag{4.3}$$

The *Z-Score* is calculated by Equation (4.3), where x_i represents an individual data-point, and μ denotes the mean and σ the standard deviation of the distribution. In detail, the *Z-Score* measures how many standard deviations a specific data point lies away from the mean of the estimated baseline distribution.

Depending on the strictness of the task, there are two well-known *Z-Score* guidelines. According to Tabachnick et al. [TFU07] *Z-Scores* that exceed 3.29 are considered outliers. Other works [Che+24; RKK24] utilize Pukelsheim's *three sigma rule* [Puk94], that means that samples with a larger distance than three σ to the mean of a distribution are deemed outliers. In a symmetric, normal distribution, a *Z-Score* of 3.29 corresponds to a probability of less than 0.001 for values beyond this threshold belonging to the distribution, which is why they are confidently considered outliers.

When comparing code similarity, it may be better to use the slightly less deviating three sigma rule [Puk94]. Using the stricter 3.29 threshold might filter out perturbations that produce strong changes, yet are similar enough for a model to expect robust results. By choosing three sigma, it is rather unlikely that meaningful robustness perturbations will be automatically classified as too different for the evaluation.

Consequently, when there is a clear correlation between *cosine similarity* and translation success, perturbations that produce similarity values below three sigma can be considered less valuable for the robustness assessment, and values above it indicate perturbations the model should definitely be robust against.

Recall that the baseline is a result of pairwise comparisons of the C files. Therefore, the value range reflects possible values for code inputs, not for basic natural language, as it would be found for the one prompt's instruction. Note that every file in the dataset has the same **Identity** instruction, so it is not possible to create an instruction baseline with the benchmark dataset. Subsequently, the baseline of the code files also has to be used for assessing instruction perturbations. Since the robustness against perturbed instructions is not relevant for the code translation system, there was no specifically created dataset for different instructions, to allow for a measured baseline of instruction *cosine similarity* values.

4.5 Application of the Framework

After the three steps of the framework have been presented, this section describes how the framework can be used methodically to perform a comprehensive robustness evaluation. This methodology builds the basis for the experiments presented in the following chapters, which are specifically tailored to the five RQs.

The herby explained methodology indirectly accounts that the components of this framework can be used to comprehensively evaluate the robustness of an LLM in code translation, which is the core RQ of this thesis. The subsequent experiments empirically show whether Step I to III present the necessary components.

Furthermore, the upcoming section presents an approach distinguishing between inherent LLM noise and true robustness deficits, which addresses RQ2: “How does one differentiate between inherent LLM nondeterminism (noise) and true robustness deficits?”

4.5.1 Differentiate Between Model Noise and Robustness Deficits

As detailed earlier, LLMs are stochastic models. Additionally, when considering the common temperature-based sampling during the output generation, it becomes clear that LLMs can produce different outputs for identical inputs. Recall that the thesis’s definition of robustness accounts for this characteristic. Specifically, a LLM-based system is deemed robust if it is able to maintain an acceptable and expected level of performance under input variations. Therefore, a crucial component of a robustness evaluation is to investigate what the expected level of performance is and what performance signals non-robust behavior, therefore allowing a distinction between non-deterministic noise and true robustness deficits caused by perturbations.

Solving this question is important for a comprehensive robustness evaluation. If an evaluation does not manage to isolate the model’s normal fluctuations, it risks attributing random variations in performance to non-robust behavior. Preventing this is the core motivation behind RQ2. To solve this challenge, the thesis introduces the following approach.

Concept

The proposed methodology establishes a statistical baseline to quantify the LLM’s inherent performance variability. This is achieved by collecting a larger number of translation results N_{total} for the unperturbed **Identity** prompts compared to the standard number of runs n used when evaluating specific perturbation instances (e.g., $N_{total} = 20$ baseline runs vs. $n = 5$ runs per perturbation). This larger N_{total} allows for a more reliable estimation of the model’s typical output distribution under normal conditions.

Specifically, the N_{total} baseline results are used to estimate the distribution of $RP_s@k$ values that one would expect if only n runs had been performed, thereby mirroring the exact evaluation conditions of the perturbation experiments. This estimated baseline distribution serves as a reference point.

In detail, such a value distribution allows defining statistical thresholds that classify whether the performance under a perturbation resulted in a performance that deviates significantly from the expected range. This is similar to the *cosine similarity* baseline approach that examines the *cosine similarity* value range. If the performance under a perturbation is clearly part of the measured unperturbed performance value range, it cannot confidently be distinguished from nondeterministic model noise, since the model showed the same deviating performance for the **Identity**.

Similar to the *cosine similarity* baseline, the methodology utilizes the *Z-Score* (Equation (4.3)) to specify significant outliers. However, while for *cosine similarity* the *three sigma rule* is used to specify outliers, classifying non-robust behavior is more strict. In the context of robustness evaluation, it may be beneficial to prioritize *precision* over *recall* [DG06]. This means minimizing the risk of falsely classifying noise as a robustness deficit (false positives), even at the cost of potentially missing some minor deficits (false negatives). Consequently, instead of using *Z-Score* threshold of 3, the robustness classification uses the stronger deviating *Z-Score* of 3.29, which originates from Tabachnick et al. [TFU07]. As detailed earlier, this threshold corresponds to $p < 0.001$ in a normal distribution, where p is the probability of a datapoint belonging to the distribution. Therefore, larger absolute *Z-Scores*[9] than this threshold are highly unlikely to be caused by random fluctuations alone and are more likely to signal a genuine robustness issue.

While the *Z-Score* approach yields a statistical reference point, it should be applied with caution. The previously presented threshold probabilities are bound to the model, producing a normally distributed performance value range. Since the explored literature suggests that this is a novel method for distinguishing between an LLM's noise and true robustness deficits, the experiments have to account for it and may adapt the approach if necessary. Since there is no ground truth or perfectly robust model one can use for comparison, this method cannot be completely validated. The specific implementation details for estimating the baseline distribution are described next.

Implementation of Baseline Distribution Estimation

This section details the practical methods used to calculate the estimated baseline distribution of $RP_s@k$ based on n runs, derived from the larger set of N_{total} available **Identity** runs.

Recall that *deterministic* perturbations are being evaluated with $RP_1@k$ and *stochastic perturbations* with $RP_{s>1}@k$. Since larger s potentially also poses a challenge for the model under **Identity** by penalizing fluctuations, it is necessary to calculate a baseline for $s = 1$ and a baseline for the s chosen for the *stochastic perturbations*.

Baseline for s=1 To estimate the baseline distribution of $RP_1@k$, all possible unique subsets of size n are drawn from the N_{total} available baseline runs. For each subset, the $RP_1@k$ is calculated. The total number of such subsets is given by the binomial coefficient $\binom{N_{total}}{n}$. For instance, with $N_{total} = 20$ baseline runs and an robustness evaluation with $n = 5$, this results in $\binom{20}{5} = 15,504$ distinct $RP_1@k$ values. The frequency of specific values forms the baseline distribution that is desired. While for this setting the $\binom{20}{5}$ values are computationally feasible, the number of computations explodes when working with larger s.

Baseline for s > 1 Calculating the exact distribution for $s > 1$ by enumerating all combinations of s-sized subsets $\binom{N_{total}}{s}$ that form n-sized subsets $\binom{\binom{N_{total}}{s}}{n}$, is computa-

[9]Absolute *Z-Score* refers to positive and negative values.

tionally infeasible. Therefore, *bootstrap sampling* [ET94] is employed to approximate the distribution.

To estimate the distribution of $RP_s@k$ based on n runs per seed, a large number of random samples denoted as b (e.g., $b = 10,000$) are generated as follows. First, generate a set of size b with randomly sampled subsets of size s. These subsets are drawn from the N_{total} translation results to form the groups considered for the worst-case measurement. Specifically, this means that a task has to be successful for each of these s groups to be considered correct.

Having this group of size s allows to repeatedly sample groups of size n (b times). Each of these n-sized groups represents a single potential $n \times s$ evaluation, which would be conducted in the robustness evaluation for *stochastic perturbations*. This generates a large set of $RP_s@k$ values that approximate the true distribution expected under the n-run evaluation for *stochastic perturbations*.

Naming the Methodology Both the combinatorial method (for $s = 1$) and bootstrap sampling (for $s > 1$) provide the necessary estimated baseline distributions (including mean and standard deviation) required for the *Z-Score* analysis described in Section 4.5.1, enabling a statistically guided differentiation between model noise and significant robustness deficits. The later experiments will refer to this technique with *Sampled Identity*. Even though it does not include "sampling" for $s = 1$, *Sampled Identity* was chosen for consistency and clarity.

4.5.2 Comprehensive Robustness Evaluation

After the baseline for an LLM has been established, the comprehensive robustness evaluation can be conducted. The same dataset used for the creation of the baseline can be perturbed with the implemented perturbation strategies in Step I and repeatedly translated with Step II. With the translation results of Step II, the components of Step III can be leveraged to perform the evaluation under specific points of view.

Aggregated Multi-Level Analysis

As detailed before, the robustness metrics can be aggregated based on different characteristics: (*i*) perturbation determinism (deterministic vs. stochastic), (*ii*) perturbation target (instruction, comments, code), or (*iii*) perturbation level (I-VI).

With such a multi-level analysis, the evaluation examines whether certain groups of perturbations cause stronger robustness deficits than others. One intuition that was described earlier was that instruction might have the potential to cause stronger deviations than code perturbations. Furthermore, it could reveal whether deeper perturbations in the code's flow pose more challenge than superficial modifications to comments or identifiers.

In case the model does not signal any clear weaknesses against these aggregations, this suggests that the model overall shows robust behavior. However, since these aggregations might hide certain robustness deficiencies, a perturbation-specific analysis should be performed as well.

Perturbation-Specific Analysis

Each perturbation strategy can be analyzed individually and compared with the baseline. The *Z-Score* suggests whether certain strategies reflect non-robust behavior or nondeterministic noise.

Perturbations that cause significant non-robust behavior could also be analyzed at the file level. By analyzing the results on the file level, it might reveal an investigation into why the model shows deviations for a specific perturbation. However, conclusions about reasons are primarily guessing, since it is hard to fully understand the model's "thoughts". Explaining the real reasons behind a model's behavior is an ongoing research area, which goes beyond the thesis's scope [Zha+24b].

Integration of the Semantic Similarity Analysis

In addition to the perturbation-specific results, the semantic similarity analysis is incorporated into the evaluation. This additional analysis offers insights into why a model might struggle with certain perturbations and verifies whether the implemented perturbations are sufficiently similar to enable a meaningful robustness assessment.

Model Comparative Analysis

By applying the same methodology across different models, their robustness properties can be systematically compared. A separate baseline must be established for each model to account for model-specific non-deterministic behavior and to serve as a reference for measuring robustness.

This comprehensive robustness evaluation methodology enables a precise, statistically referenced analysis of the robustness of an LLM-based code translation system and presents all information needed to empirically answer the RQs in the final Chapter 10.

5 Experimental Setup and Baseline Analysis

This chapter lays the groundwork for evaluating the code translation system featuring *GPT-4o-mini*. It establishes the baseline performance of the default translation pipeline, which utilizes *feedback loops* in *Step II* of the framework (Section 4.3). Understanding this baseline is crucial before assessing the system's robustness against perturbations in the subsequent chapters.

To enable a meaningful interpretation of robustness results later, this chapter first examines the baseline performance of the code translation system without any perturbations. This baseline aims to capture the inherent fluctuation of the system. Quantifying this noise is essential for assessing the system's robustness later and differentiating between inherent model variability and true non-robust behavior when perturbations are applied. This analysis provides the foundation for addressing RQ2 ("How does one differentiate between inherent LLM nondeterminism (noise) and true robustness deficits?").

Before examining the baseline, the chapter explains the experimental setup, including the dataset that is used for the robustness evaluation across the thesis. Furthermore, the utilized configurations of the perturbation strategies are presented.

5.1 Experimental Setup

This section explains the preliminaries that are necessary to perform and evaluate robustness experiments. That includes explaining the benchmark dataset, which was used as input to the proposed framework, and also the evaluated LLMs and their parameters. Furthermore, the section details the applied perturbation strategies and their configurations.

5.1.1 Benchmark Dataset

All experiments were conducted on the same benchmark dataset. This section presents a profound investigation of the dataset to enable a detailed interpretation of the experiment results.

The dataset was prepared by Bosch and has been used in previous in-house code translation evaluations with their designed code translation system explained in Section 4.3.4. As mentioned in Section 4.2.4, a well-defined dataset is essential for a thorough evaluation.

The dataset consists of 50 *C* files that include 76 functions. 30 of these files are internal automotive embedded code that were extracted from real-world Bosch projects [HQS24]. Using such real-world code allows an honest evaluation of the translation system, because *data leakage* is minimized. While proprietary code does not ensure that the model has never been trained on similar files, the chances of *data leakage* are reduced, in comparison to open-source projects or well-known code generation benchmarks. Moreover, these files not only present simple coding interview-style code, but also relevant real-world code. Specifically, this code includes functions from different domains and layers in the *AUTOSAR Classic* architecture [AUT]. That involves low-level code, base software library code, as well as application-level control code. Most of it is hand-written code, but some files also contain auto-generated code.

In addition, the dataset includes ten files of open-source code that have been used in prior LLM-based code translation work [Eni+24]. This open-source code originates from an audio processing and sound card emulation library. The last ten files stem from competitive programming solutions that were also used in prior works [Yan+24b; Sza+23]. This data was scraped for unsupervised learning and is thoroughly explained by the papers. While the origin is interesting for the interpretability of results, there are other key factors that can be characterized. For LLM-based *C* to *Rust* translation, two key factors must be considered: the number of tokens per file and code features.

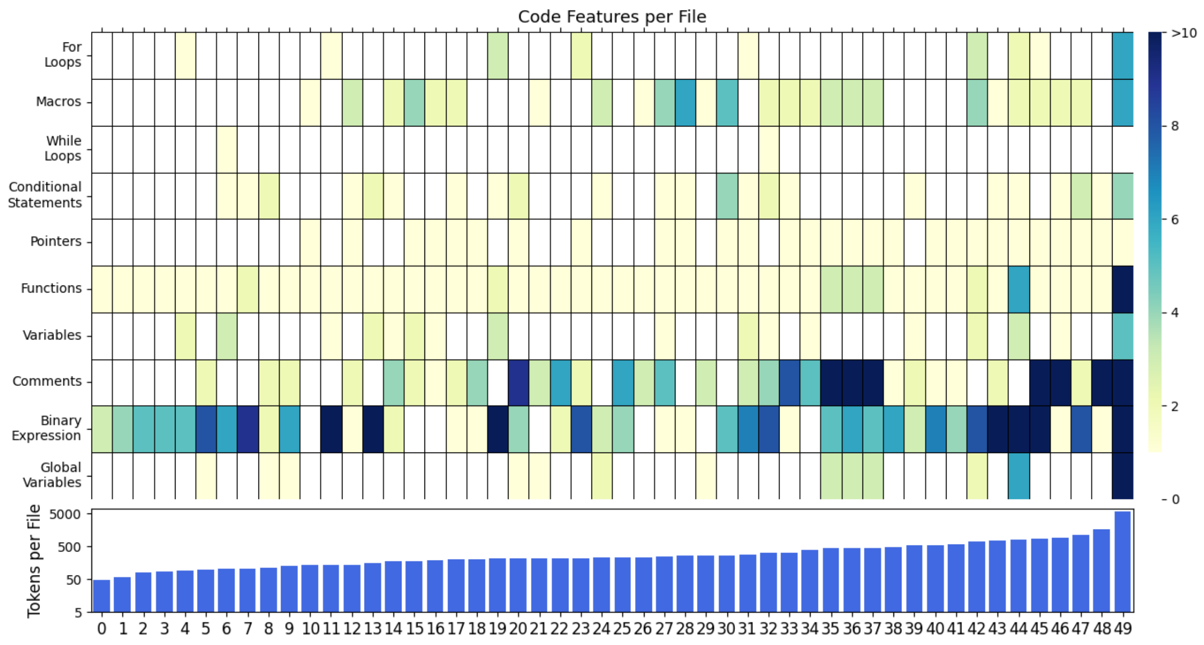


Figure 5.1: Distribution of code features and file sizes among the benchmark dataset. The heatmap visualizes the occurrence of different code features across all files. Darker colors indicate a higher frequency of the respective feature. The bar plot below represents the file sizes in terms of token count on a logarithmic scale.

Number of Tokens per File

As previous works highlighted, the number of input tokens has a big influence on the model's translation performance [Eni+24; Yan+24b; PBY24]. Thus, the dataset has to include a range from small to large files, which consequently correlates to the number of tokens. Figure 5.1 illustrates that the dataset covers a diverse range of tokens per file, when tokenized with *tiktoken*[1]. Most files have a token count between 50 and 1000, which represents files with 190 to 3400 characters and 10 to 110 lines of code. So the majority of files represent rather small files, following a single responsibility [Mar14]. However, files 47 to 49 present files containing 1142-5716 tokens, which most likely results in increased translation difficulty for these files.

[1]This is the tokenizer that OpenAI uses [Ope25c].

Code Features

While the number of tokens influences a model's translation performance, the characteristics of the source code's features are also a relevant part of the investigation. Specifically, as explained in Section 2.1, there are some features in *C* that are not trivially translatable to *Rust*.

A *for-loop* or a *binary-expression* can most likely be easily converted to *Rust*, as it also includes loops and parenthesized expressions. However, *Rust* does not directly allow the declaration of global variables or macros [PG24; Cai+25]. Furthermore, *Rust* is more restricted in the context of pointers.

As shown in the heatmap in Figure 5.1 the dataset includes simple and complex features. This enables the robustness evaluation of whether the model's ability to generate alternative, idiomatic solutions for not native *Rust* features is influenced by perturbations.

5.1.2 LLMs and Parameters

The experiments of this thesis utilize LLMs accessible via API and locally running LLMs. This section briefly explains how the LLMs were chosen, highlighting the trade-off between cost and inference time.

Azure OpenAI

Specifically, the LLMs are provided by an *Azure* Instance [Mic25]. This service hosts SOTA LLMs by *OpenAI*, such as *GPT-o1* [Ope24c], *GPT-4o*[Ope24b] & *GPT-4o-mini* [Ope24a], or *GPT-3.5-Turbo* [Ope23].

Model	Input Token Cost (1M Tokens)	Output Token Cost (1M Tokens)
GPT-4o-mini	$0,165	$0,66
GPT-3.5-turbo	$1,50	$2
GPT-4o	$2,75	$11
GPT-o1	$16,50	$66

Table 5.1: Comparison of available LLMs on *Azure* regarding price per token, based on [Mic].

Table 5.1 illustrates the cost differences between these LLMs. Since the robustness evaluation requires a large number of LLM-calls (see Equation (4.1)), not all available models could be included, as this would have been too expensive. Therefore, the decision was to focus the evaluation on *GPT-4o-mini*. This model is in widespread use due to its interesting cost-performance and availability characteristics, and therefore also is the default LLM of the code translation system. Additionally, a deterministic perturbation experiment with *GPT-3.5-turbo* is feasible. This is an old, powerful model that was used extensively in previous work.

Local Models

Besides the *Azure* hosted LLMs, the thesis also had access to locally running LLMs. Although these models do not produce direct costs aside from hardware operation, their execution time on the dataset is longer, as the models' inference was not on optimized hardware.

The recent model *Phi-4* [Abd+24] demonstrated acceptable runtime performance, allowing the examination of the model's robustness under *deterministic* strategies, with $s = 1$ and $k = 5$. The compact size and efficiency of *Phi-4*, combined with its open-source availability, made it suitable for local execution despite hardware limitations. However, since the experiment for *stochastic* perturbations incorporates $s = 3$, the number of runs is significantly higher than for the *deterministic* perturbations, which prevented the examination of *Phi-4* for this experiment.

As all models are examples of *general purpose* LLMs, another open model specifically designed for code generation was evaluated, namely the 14 billion parameter version of *Qwen2.5-Coder* [Hui+24]. In the subsequent sections and chapters *Qwen2-5-Coder-14B* is consistently denoted as *Qwen2.5-Coder* because of clarity. The decision to use the *14B* version was because it was the largest version of *Qwen2.5-Coder* that worked on the used hardware for the experiments. Moreover, the next section will show that this version of *Qwen2.5-Coder* will have approximately the same number of parameters as *Phi-4*, which enables a fairer comparison. Nonetheless, the computational execution of this model posed even greater challenges for the hardware compared to *Phi-4*. This resulted in only evaluating the robustness for deterministic strategies with $n = 1$, and subsequently preventing *stochastic* perturbation evaluations. This limitation reduces the interpretability of the results. However, as this would be a chance to evaluate a model exclusively trained for code generation, the decision was made to perform at least one run.

LLM Settings

Unlike *ReCode* [Wan+23], all models in this thesis are used with their default parameters, meaning no *greedy-sampling* is applied.

The major goal of this thesis is to provide a robustness evaluation for LLM-based code translation in a real-world environment. Since the evaluated code translation system operates with default parameter settings, no modifications were made, as variations in temperature values and sampling methods are known to significantly impact performance [Liu+23]. In detail, the temperature is set to 0.7 by default for all models.

Furthermore, for a better interpretation, the size of the relevant models should be mentioned (see Table 5.2). While the model sizes are known for the local models *Phi-4* [Abd+25] and *Qwen2.5-Coder* [Hui+25], the number of parameters for the *GPT* models has never been officially published. However, a recent paper by *Mircosoft* [Aba+24] estimated the model size of *GPT-4o-mini*, and has since then been commonly taken as a reference. For *GPT-3.5-turbo*, multiple sources mention twenty billion parameters. This goes back to this *Forbes* article [Far17], and was also referenced in a paper by *Microsoft* [Sin+23], which was later withdrawn and released without giving information about *GPT-3.5-turbo*'s parameter size. So the given approximation for *GPT-3.5-turbo* is very uncertain.

Model	Number of Parameters
GPT-4o-mini	≈ 8 billion
Phi-4	14 billion
Qwen2.5-Coder	14.7 billion
GPT-3.5-turbo	≈ 20 billion

Table 5.2: Comparison of number of parameters of the relevant LLMs.

5.1.3 Perturbation Configurations

As explained in Section 4.2 *Step I* of the framework performs the perturbations based on a configuration. This section details the applied perturbation strategies and utilizes embedding models to illustrate the similarity between perturbed prompts.

To address the RQs, this thesis defines two configurations: one for *deterministic* and one for *stochastic* perturbations.

Deterministic Perturbations

The *deterministic* configuration utilized 12 perturbation strategies, which resulted in 20 perturbed datasets, including one **Identity** dataset. The Strategies **Backtranslation**, **CodeFormat**, and **Translation** were performed with multiple parameters, which explains why there are more perturbation datasets than strategies. The distinct perturbations are included in Table 4.1 and Table A.1.

Specifically, there are three perturbations on the instruction, four on comments, and 14 perturbations targeting code.

Perturbations Targeting Comments and Code The perturbations targeting code and comments are visualized in Figure 5.2.

Specifically, the figure visualizes embeddings generated by an *OpenAI* embedding model[2] for all files across all perturbation datasets, reduced in dimensionality using *Uniform Manifold Approximation and Projection* (UMAP) [MH18].

This visualization represents file similarities. Files positioned closer together are considered more similar by the embedding model.

The plot indicates that perturbation strategies produce highly similar files, as points of the same color are clustered together. This aligns with the requirement of semantical equivalence for perturbations.

Only the **ABC** perturbation appears to introduce greater variance, as its points are more distant from the belonging **Identity** for most files.

However, this does not inevitably mean that **ABC** produces very different files. **ABC** is a perturbation that results in all files having identical identifier names. Consequently, this leads to each **ABC** file being also similar to the other **ABC** files, which results in the points not only being drawn to the **Identity** of the same file, but also to other **ABC** perturbations.

A closer examination of the **Identity** points reveals that files 35, 36, and 37 (UMAP Dimension 1: 12.5-15, Dimension 2: 25) are very close to each other. Figure 5.1 further confirms this, as files 35, 36, and 37 exhibit similar code features and token lengths.

The alignment between Figure 5.1 and the UMAP plot, along with the clear clustering between other files, suggests that the embedding model effectively captures differences in similarity.

When incorporating the magnitude of perturbation change into the robustness evaluation, this is important to keep in mind.

Perturbations Targeting the Instruction Figure 5.2 only visualized perturbations on the code part. Hence Figure A.1 visualizes the perturbations targeting the instruction, to illustrate all deterministic perturbations.

[2]The exact model is *text-embedding-ada-002* [Ope]

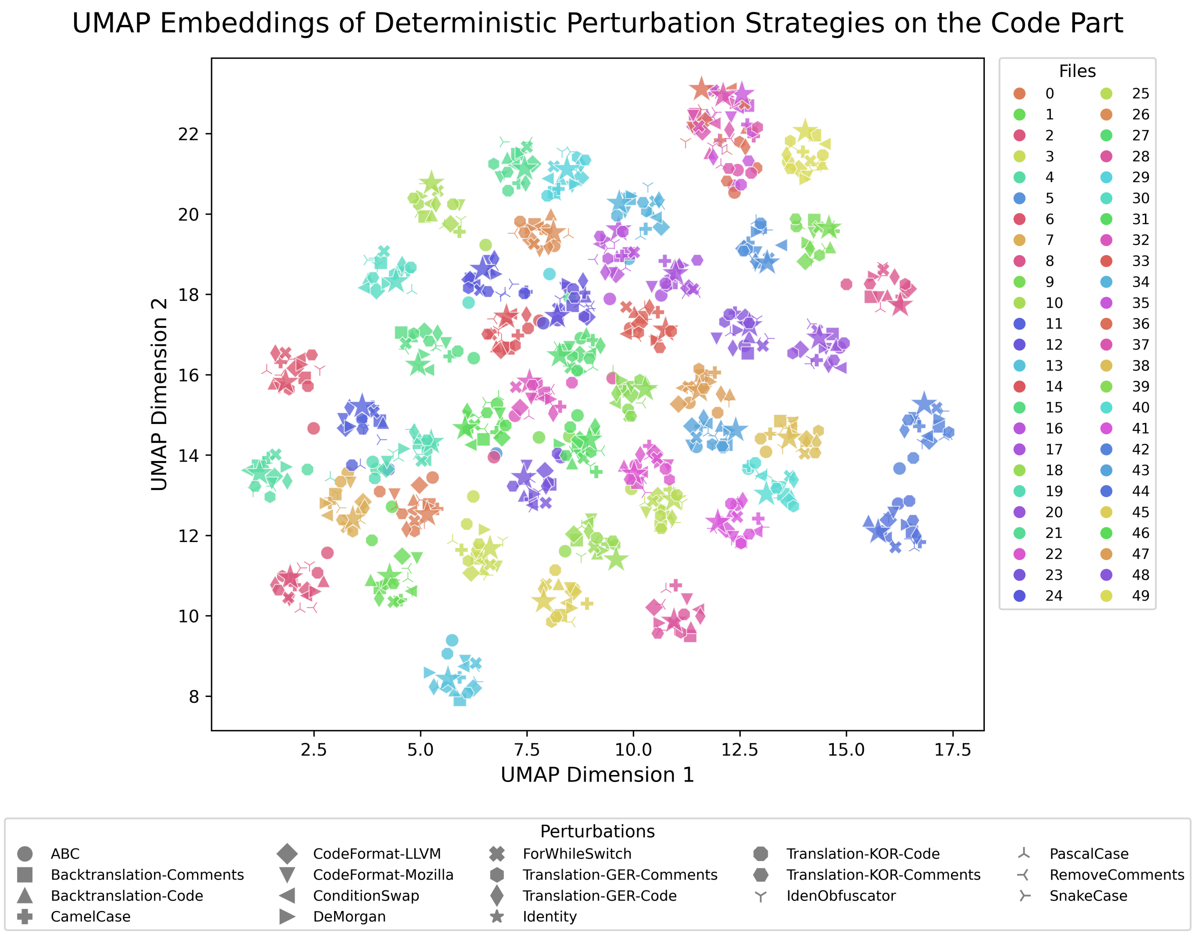


Figure 5.2: UMAP embeddings of the 50 color-coded *code* files, each subjected to multiple *deterministic perturbation strategies* indicated by different markers. The plot shows that the different files still form clusters with their perturbations, suggesting that the perturbation strategies produced semantically similar *code*.

Since the perturbed instructions are identical for all files among the same perturbation strategies, this plot is included in the Appendix.

Stochastic Perturbations

The configuration for the *stochastic* perturbations utilized 11 strategies and **Identity** that resulted in 16 perturbation datasets (48 when considering $s = 3$), four instruction perturbations, two perturbations on comments, and ten on code.

Specifically, **Butterfinger** and **ChangeCharCase** were performed on all three different perturbation targets, resulting in more datasets than *stochastic* perturbation strategies.

Perturbations Targeting Comments and Code Figure 5.3 visualizes the embeddings of the perturbations with UMAP, similar to the figure for *deterministic* strategies.

The plot also shows that the perturbations produced code similar to the **Identity**, as the different files are mostly separated by their position.

However, there is a cluster in the center of the graphic that shows an overlap of **Dead-CodeInsertion** for different colors. This is explainable because the **DeadCodeInsertion**

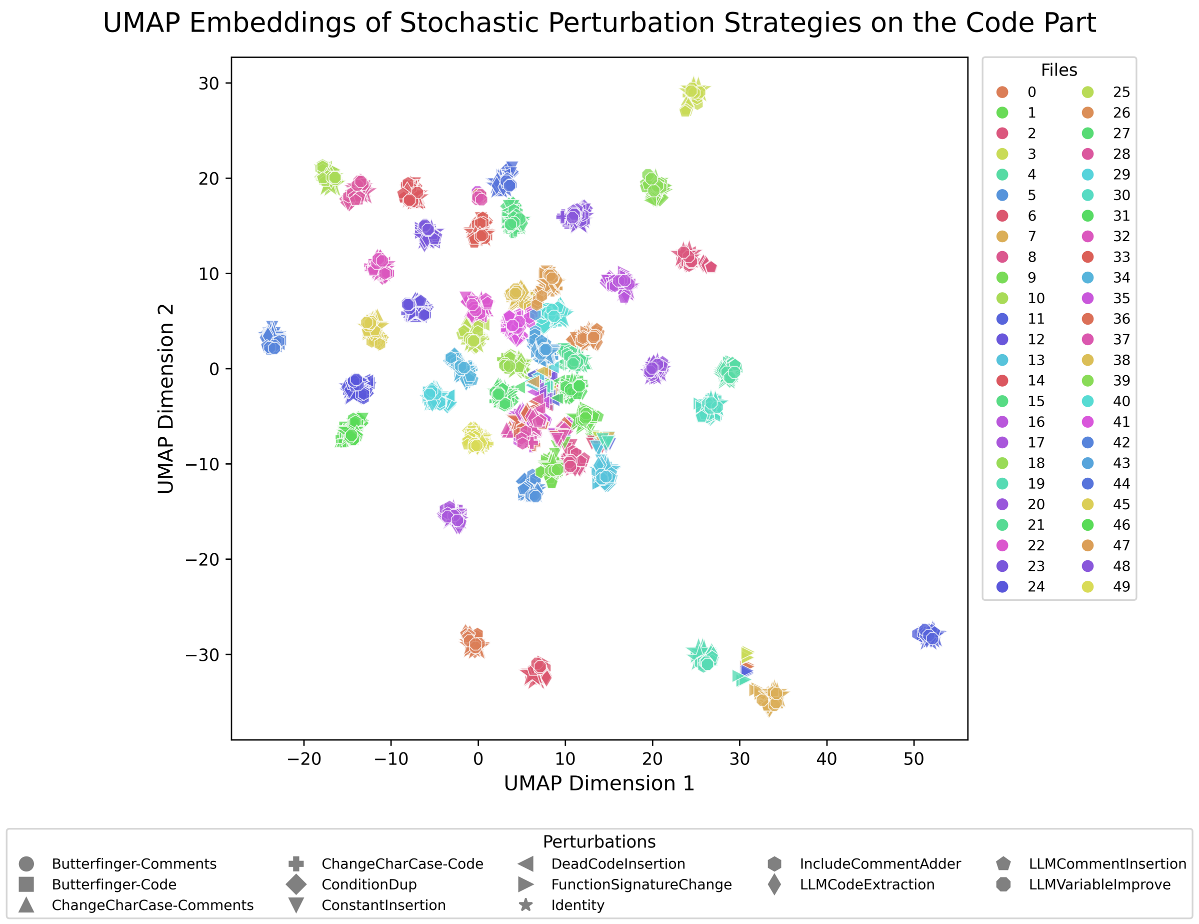


Figure 5.3: UMAP embeddings of the 50 color-coded code files, each subjected to multiple *stochastic perturbation strategies* indicated by different markers. The plot shows that the different files still form clusters with their perturbations, suggesting that the perturbation strategies produced semantically similar *code*.

adds predefined code snippets to the code. The more of these snippets that can be added to a file, the more similar it will get to another file that has a large ratio of inserted dead code snippets. Therefore, it is reasonable that some files have overlapping embeddings. This does not necessarily imply a violation of the semantic similarity requirement, instead, files with extensive dead code insertions are more similar to other extensive dead code perturbed files than to their original **Identity**. The *cosine similarity* to the **Identity** may not be large, which is investigated in Chapter 8.

The same can be found in the bottom right, for **FunctionSignatureChange**, which follows the same explanation as for the overlap of **DeadCodeInsertion**. In contrast to **DeadCodeInsertion**, **FunctionSignatureChange** adds predefined parameters to function definitions, making the perturbed code more similar to each other.

Perturbations targeting the Instruction Figure 5.4 visualizes the embedding UMAP of the stochastic perturbations on instructions.

The plot shows that all perturbation strategies are more similar to each other than the **Identity**. However, some **Butterfinger** points appear close to the **Identity**, likely due to the low probability (5%) of a character replacement. There may have been cases where

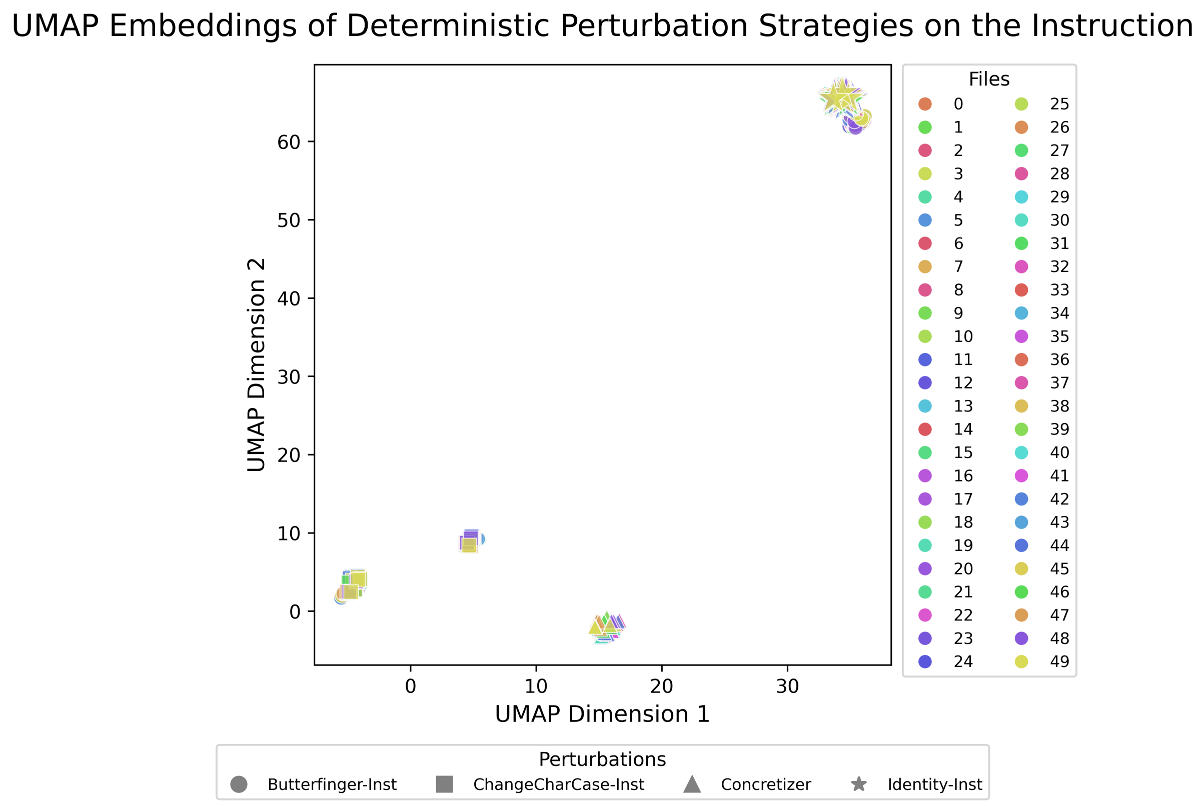


Figure 5.4: UMAP embeddings of the 50 color-coded files' instructions, each subjected to multiple *stochastic perturbation strategies* indicated by different markers. The plot shows that the different files still form clusters with their perturbations, suggesting that the perturbation strategies produced semantically similar *instructions*.

the **Butterfinger** did not produce a single change.

Whether the great distance to the **Identity** describes that the perturbations produced highly semantically different prompts remains to be investigated in Chapter 8. Recall that UMAP shows the relation between all files and not directly the magnitude of difference with respect to the **Identity**.

It can also be found that **Butterfinger** is overlapping with **ChangeCharCase**, suggesting that the model detects that the produced perturbations are an artifact of noise, and may be closely related.

5.2 Baseline Performance and Model Noise

This section establishes the normal performance characteristics of *GPT-4o-mini* with *feedback loops*, when translating code without any input perturbations. It quantifies the model's expected success rates and inherent output variability, providing essential knowledge for interpreting the robustness experiments presented in subsequent chapters.

5.2.1 Motivation

Before assessing the impact of various perturbations, it is crucial to understand how the model behaves under ideal conditions. The inherent nondeterminism of LLMs can cause performance fluctuations even with identical inputs. This experiment aims to quantify this baseline variability for *GPT-4o-mini*. Establishing this baseline is necessary for addressing RQ2 ("How does one differentiate between inherent LLM nondeterminism (noise) and true robustness deficits?")

Solving this question is important for a comprehensive robustness evaluation. If we cannot isolate the model's normal fluctuations, we risk attributing random noise as non-robust behavior. By quantifying *GPT-4o-mini*'s performance variability under unchanged conditions, this baseline experiment lays the groundwork for accurately assessing whether observed performance drops in subsequent experiments are due to perturbations or simply stochastic noise.

The underlying methodology for this differentiation is detailed in Section 4.5.1.

5.2.2 Obtaining Baseline Data

This baseline analysis utilizes data generated using the **Identity** perturbation, meaning the original *C* files and the standard translation instructions without modification were used as input to the translation system.

The **Identity** perturbation serves as a baseline condition and is included in both the *deterministic* and *stochastic* perturbation experiments detailed in Chapter 6. The *deterministic* experiments contribute $n = 5$ runs for **Identity**, and the stochastic experiments contribute $s = 3 \times n = 5 = 15$ runs (one set of 5 for each seed). This results in a N_{total} of 20 available runs for the unperturbed files.

Therefore, without performing dedicated experiments solely for the baseline, these 20 existing runs are leveraged to establish the baseline performance and estimate the model's inherent fluctuations when evaluating with $RP_s@k$.

The choice of $n = 5$ repetitions per perturbation represents a trade-off. While more runs would yield statistically more stable estimates, each additional run significantly impacts the total number of LLM calls, increasing costs and experiment runtime. Thus, $n = 5$ was deemed a practical balance between a meaningful estimation and resource, as well as time constraints for the thesis.

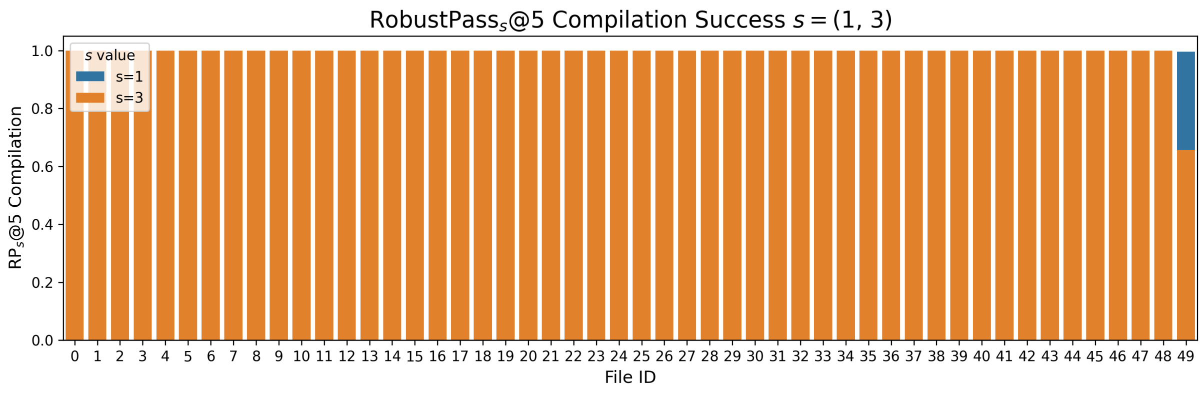
5.2.3 Baseline Performance of GPT-4o-mini

Before quantifying the noise, the regular baseline performance of *GPT-4o-mini* on the **Identity** is examined. As noted previously, the later robustness experiments on *GPT-4o-mini* yield 20 runs for the **Identity** dataset, where each run measures *compilation success* and *fuzzing success*.

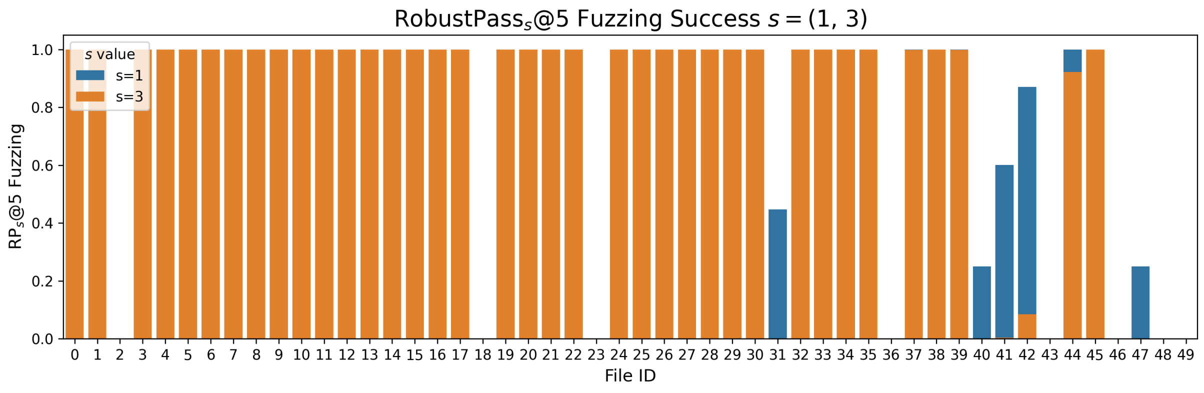
Using the defined concept, there are $\binom{20}{5} = 15504$ values for each $1 \leq k \leq 5$ of $RP_1@k$. Furthermore there are 10,000 bootstrapped samples of the $\binom{\binom{20}{3}}{5} = 1.6 \times 10^{13}$ $RP_3@k$ values for each $1 \leq k \leq 5$.

Figure 5.5 visualizes the mean $RP_s@5$ among these values for *compilation success* and *fuzzing success*. Specifically, the $RP_3@5$ (orange) bars are in front of the $RP_1@5$ (blue) bars, which is done because the mean of $RP_1@5 \geq RP_3@5$ for all files.

Since the later experiments mostly focus on $k = 5$ the the evaluation of *GPT-4o-mini*'s baseline performance also focuses on $k = 5$. Furthermore, when having $n = k$ the $RP_s@k$ values can either be 1 or 0. As long as at least one sample of five runs is correct, the probability of drawing this in $k = 5$ samples is 1. This enables an easier and



(a) Barplot for RP_1@5 and RP_3@5 on *compilation success*.



(b) Barplot for RP_1@5 and RP_3@5 on *fuzzing success*.

Figure 5.5: Barplots showing the particular file translation results with *GPT-4o-mini* utilizing RP$_s$@k for *compilation success* and *fuzzing success*. RP_3@5 (orange) is in front of RP_1@5 (blue), as the latter one is always at least as good as RP_3@5.

more explainable interpretation. Nevertheless, Appendix A.3 briefly discusses the baseline performance for $1 \leq k \leq 5$.

Baseline for Compilation Success

Figure 5.5a illustrates the mean translation correctness with respect to *compilation success* (y-axis) for all files in the dataset (x-axis). For $s = 1$, all files of the dataset were successfully translated into compiling *Rust* code. Specifically, aligned with the definition of *pass@k*, this means that the probability of generating at least one compiling *Rust* code within the five runs is ≈ 100% for each file. A close examination of Figure 5.5a reveals that the blue bar for file 49 is not exactly at 1.0, but rather around 0.996. In addition, Table 5.3 shows a standard deviation of 0.001, indicating that there is very little variation among the results, which is likely due to file 49.

When applying the stricter metric with $s = 3$, the general observation remains similar, although the difference for file 49 becomes more significant, with the probability dropping to 66.33%. This aligns with the expectation that $RP_s@k$ becomes more challenging as s increases. Furthermore, file 49 has the highest token count (Figure 5.1). The drop to 66.33% supports related findings [Eni+24; Yan+24b; PBY24] indicating that larger code segments can reduce translation performance.

Overall *GPT-4o-mini* demonstrates strong performance in translating the *C* files into compiling *Rust* code, even under the stricter $RP3@5$. This is important because an already high baseline for *compilation success* makes subsequent robustness experiments more revealing. Perturbations that significantly reduce the compilation rates thereby indicate non-robust behavior.

However, *compilation success* alone is insufficient for fully assessing the code translation capability of a model. A successfully compiled file does not necessarily indicate correct translation semantics, which is crucial when translating code. The *fuzzing success* metric explores whether the compiled code behaves as intended.

	Compilation Success	Fuzzing Success
$RP_1@5$	1.0 ± 0.001	0.787 ± 0.017
$RP_3@5$	0.993 ± 0.009	0.739 ± 0.007

Table 5.3: Baseline mean ± standard deviation translation success with *GPT-4o-mini* among all files utilizing $RP_s@k$ for *compilation success* and *fuzzing success*.

Baseline for Fuzzing Success

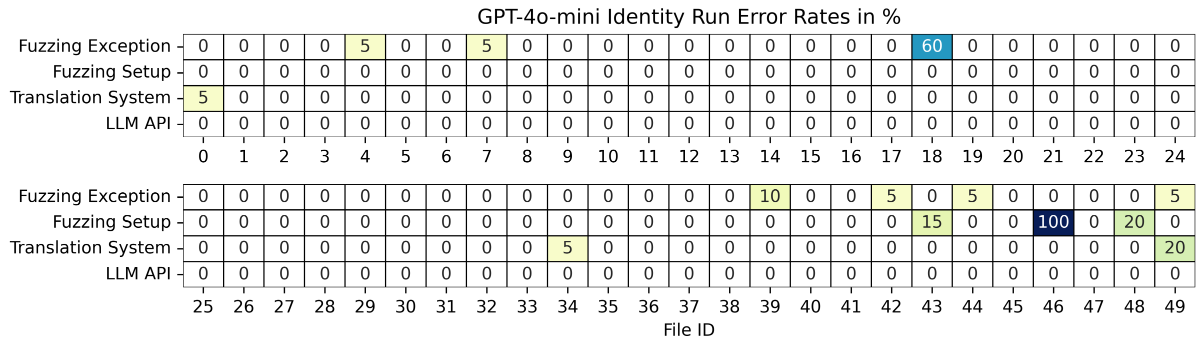
Table 5.3 reveals that translating into semantically equivalent *Rust* code is indeed more challenging than achieving *compilation success*. For $RP_1@5$, *GPT-4o-mini* yields a mean of 78.7% across all files, with a higher standard deviation of 0.017 compared to *compilation success*'s variation. Figure 5.5b illustrates how these $RP_s@5$ results (y-axis) vary among different files (x-axis), which can be grouped into three categories *perfect*, *incorrect*, and *variational*.

perfect For a total of 37 files *GPT-4o-mini* achieves a mean $RP_1@5$ of around 100%. Increasing s, affects only three of these files (i.e., 37, 39, and 44), with 44 experiencing the only noteworthy drop (0.08).

Consequently, even with $s = 3$, the *fuzzing success* for this group remains strong. These 37 files represent 74% of the dataset, which is a good performance overall. However, there are also files that, on average, never produce semantically equivalent results.

incorrect Files 2, 18, 23, 36, 43, 46, 48, and 49 show a mean $RP_1@5$ of 0.0. While this also correlates with the higher difficulty for increasing token counts, the files 2, 18, 23, and 36 are surprising outliers, given that they are surrounded by *perfect* translations and the x-axis as well as the file ID is sorted by token count.

Hence, these failures cannot simply be explained by the token count. Earlier, Section 5.1.1 discussed potentially challenging code features for a translation from *C* to *Rust*. However, comparing the features of the surprisingly failing files in Figure 5.1, does not suggest a feature that introduces such a failure. Recalling Figure 5.2, it is even more surprising that 36 failed. The files 35, 36, and 37 have very similar token counts, code features, and are also very closely related according to the embedding. 35 and 37 show perfect translation results, whereas 36 failed completely. This indicates that the failing *fuzzing success* may result from highly individual or subtle aspects.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Fuzzing Exception	0	0	0	0	5	0	0	5	0	0	0	0	0	0	0	0	0	0	60	0	0	0	0	0	0
Fuzzing Setup	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Translation System	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LLM API	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
Fuzzing Exception	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	5	0	5	0	0	0	0	5
Fuzzing Setup	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	100	0	20	0
Translation System	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20
LLM API	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.6: Heatmap showing error rates in % for the *GPT-4o-mini* **Identity** runs across all files. *Fuzzing Exception* denotes unforeseen failures of the differential fuzzing mechanism. *Fuzzing Setup* errors describe that the model generated not the same function names as in the *C* code. *Translation System* summarizes problems in the evaluated code translation system. *LLM API* includes both *Internal Server Error* [Mozb] or *Bad Request* [Moza] that showed in some experiments.

As mentioned in Section 4.3.4 *differential fuzzing* is an experimental process and can produce unexpected errors. Similarly, the tested code translation system in general, and the API service, can also produce unforeseen inconsistencies. Figure 5.6 presents the encountered error rates for each file among all **Identity** runs. Specifically, the error rate describes the percentage of all runs for a file in which an error occurred. Consequently, a 50% error rate would mean that such an error occurred in ten of the 20 runs for the file.

Comparing the error rates per file in Figure 5.6 and the *fuzzing success* in Figure 5.5b highlights that error rates between 5% – 10% do not necessarily affect the performance when evaluating with $n = k = 5$. Specifically, the files 0, 4, 7, 34, or 39 have error rates of 5-10%, nonetheless they resulted in perfectly translated files. This is because $RP_s@k$ returns 1 as long as one run resulted in a correct output. Hence, there have to be at least five failing runs (25% of 20 runs) to prevent a group from succeeding completely. As the mean $RP_s@k$ is calculated over all $\binom{20}{5}$ groups of five, such an effect would then show only in one single group. Nevertheless, in this case, there would be other groups that contain errors, and the chance of involving a correct translation is decreased with every failing run, but yet such error rates cannot directly explain *incorrect* files.

That highlights that $k = 5$ is an appropriate value for the robustness evaluation, as it is not sensitive to minor inconsistencies produced by the experimental design of the code translation system and proposed framework. Therefore, perturbations that heavily influence the model's performance show meaningful robustness deficiencies.

Other than that, file 18 involved *fuzzing exceptions* in 60% of the runs and yet Figure 5.5b shows a mean $RP_s@5$ of 0. That means that the *fuzzing exceptions* alone are not the reason for the file failing, but also incorrect translations. Furthermore, a similar behavior can be observed for file 43 that produced *fuzzing setup* errors in 15% of runs, yet resulted in $RP_s@5$ of 0.0. The only *incorrect* file that is directly and unquestionably caused by errors is 46, which produces *fuzzing setup* errors in all of its runs. However, these errors describe the model not accurately translating the function names, such that the *differential fuzzer* cannot create one-to-one comparisons. This behavior can be deemed as an incorrect translation, meaning that a $RP_s@5$ of 0.0 accurately represents the model's performance for this file. Lastly, files 48 and 49 involve error rates of 20%, or 25%, while these higher could in theory slightly influence the RP_s@5, this cannot explain files having a zero probability of being translated correctly.

This highlights again that failures stem from highly individual factors.

variational Files 31, 40 – 42, and 47 have RP_1@5 values ranging between 25% and 81%. With $s = 3$, their means drop drastically. This is an expected behavior, if the model's outputs vary from run to run, it becomes increasingly unlikely to produce three correct solutions in separate attempts ($s = 3$).

Considering this, the *perfect* files are even more meaningful for the robustness evaluation, since this really means that they have been consistently translated semantically correctly.

Furthermore, error rates can be excluded as a reason for *variational* results. The only *variational* file with errors is 42, and the error rate only refers to one single run, which, by definition of the experiment, can only have a small effect on the metric.

Overall, the baseline translation performance of *GPT-4o-mini* is not perfect with the used code translation system. However, the results are very well suited for a robustness evaluation. Subsequent experiments will demonstrate whether the model can still achieve 74% *perfect* and consistent results under perturbations. Furthermore, it is possible that perturbations reveal improved performance for *variational* or *incorrect* files. Hence, these experiments may not only show a performance drop but also a performance increase, which, according to the definition, would both be non-robust.

Before examining this, the baseline experiment has yet to show the inherent model noise. While Figure 5.5 might seem quite consistent, these results represent the mean over a very large number of potential RP_s@5 measurements. To accurately exclude nondeterminism, this is not sufficient, as a seemingly non-robust result under a perturbation for $n = k = 5$ could also be a snapshot of the model producing a result worse than the mean performance for **Identity**.

5.2.4 Noise Analysis of GPT-4o-mini

This analysis should present the distribution of RP$_s$@k values to show potential performance variations on identical inputs. Perturbations that represent outliers to the distribution are more likely to be caused by the effect of the perturbation.

Again, the evaluation is conducted for both scenarios $RP_1@k$ and $RP_3@k$. Furthermore, the subsequent sections discuss the fluctuations for $k = 5$, as this aligns with the previously examined baseline performance. When evaluating the robustness with different k values, the adequate noise measurement has to be taken into account.

Baseline for Deterministic Perturbation Strategies

The baseline for deterministic perturbation strategies is measured for RP_1@5. Figure 5.7 illustrates, that the variation for *compilation success* is negligible whereas for *fuzzing success* there are noteworthy fluctuations. By calculating all possible $\binom{20}{5}$ subsets of $RP_1@k$ *compilation success* values, it becomes clear that almost every group produces values of RP_1@5 of 100%. There is only a tiny bar at 0.98, which is so tiny that it does not affect the mean with three decimal places. This probably is the small variance that is produced by file 49. As stated earlier, the file was the only one that produced an RP_1@5 smaller than 1.0.

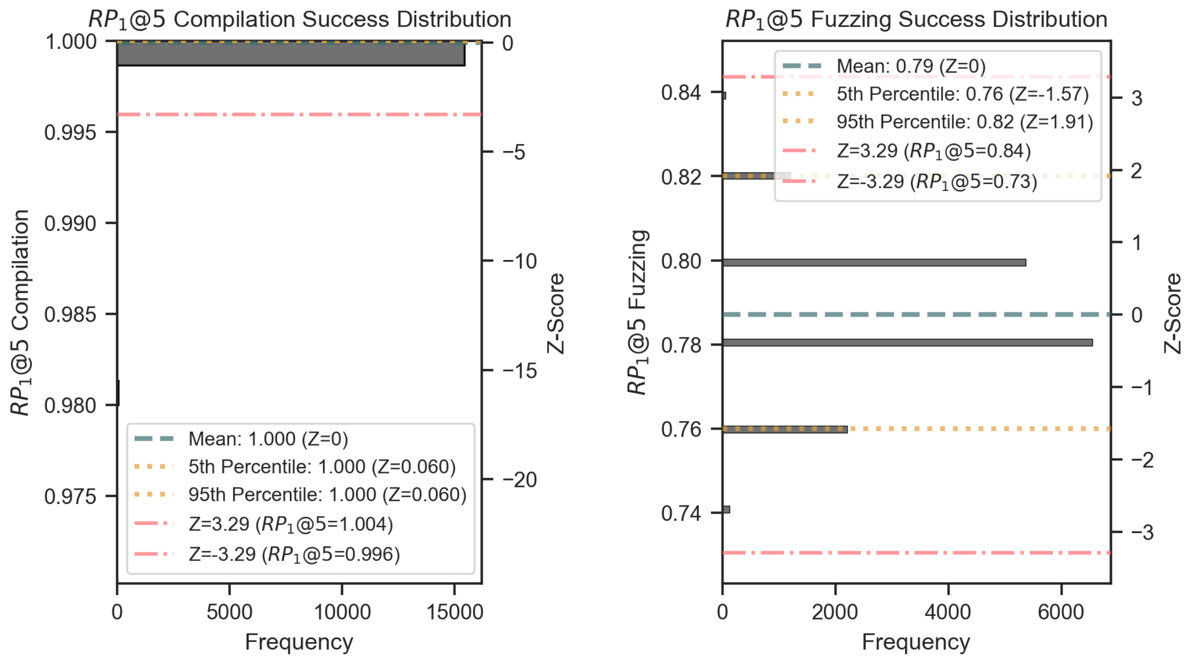


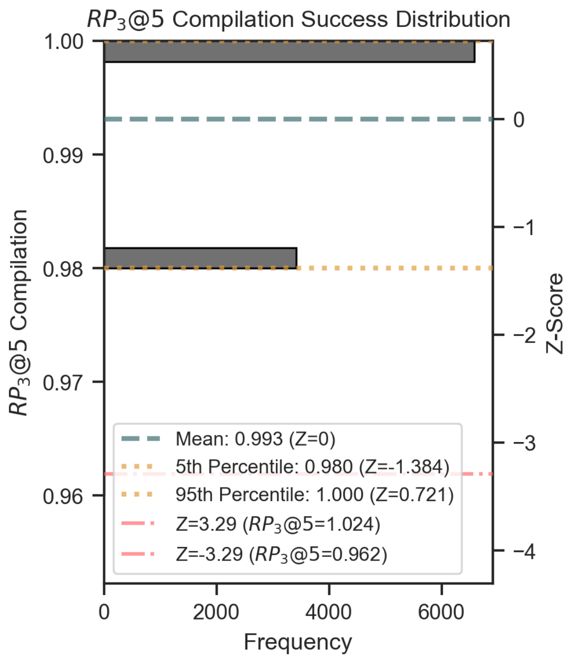
Figure 5.7: Horizontal Histograms illustrating the distribution of *GPT-4o-mini* $\boldsymbol{RP_1}$@5 on *compilation success* (left) and *fuzzing success* (right). The x-axis indicates the frequency of $\boldsymbol{RP_1}$@5 values, while the y-axis shows the actual $\boldsymbol{RP_1}$@5 value. Dashed lines mark the mean (gray), the 5th and 95th percentile (yellow), as well as the *Z-Score* of 3.29. The right y-axis shows the equivalent *Z-Score*.

Fuzzing success involves more fluctuations, which are most likely due to the *variational* files. The model did not consistently produce good results for these files, which has an effect on the average dataset performance. In the worst case, the $\boldsymbol{RP_1}$@5 might be calculated for a subset that had bad results for *variational* files. However, overall, the fluctuation is quite balanced. The upper threshold is at $\boldsymbol{RP_3}$@5 : 0.84, the lower at $\boldsymbol{RP_3}$@5 : 0.73, whereas the mean is at $\boldsymbol{RP_3}$@5 : 0.79. The distribution clearly shows that results with an absolute *Z-Score* > 3.29 are very likely not to be nondeterministic noise, as the distribution is quite sharp.

Baseline for Stochastic Perturbation Strategies

The baseline for stochastic perturbation strategies is measured for $\boldsymbol{RP_3}$@5. Figure 5.8 presents that there are minor fluctuations for *compilation success* and *fuzzing success*. As Figure 5.5a showed already, file 49 is variational for $s = 3$, which explains why in some bootstrap samples a $\boldsymbol{RP_3}$@5 of 0.98, and in some 1.0 is calculated. Comparing the distribution for $s = 1$ and $s = 3$ supports this finding. However, with a *Z-Score* of 3.29 at 0.962, the model still produced consistent output.

Furthermore, Figure 5.8 illustrates that the fluctuation for *fuzzing success* decreases with $s = 3$, compared to $s = 1$. This is an explainable behavior, as the *variational* files for $s = 1$ mostly become *incorrect* files for $s = 3$. The fewer *variational* files, the more consistent the dataset performance. Consequently, stochastic perturbation results are less prone to nondeterministic fluctuations. Perturbations that produce $\boldsymbol{RP_3}$@5 ≤ 0.716 or $\boldsymbol{RP_3}$@5 ≥ 0.763 can be considered non-robust.



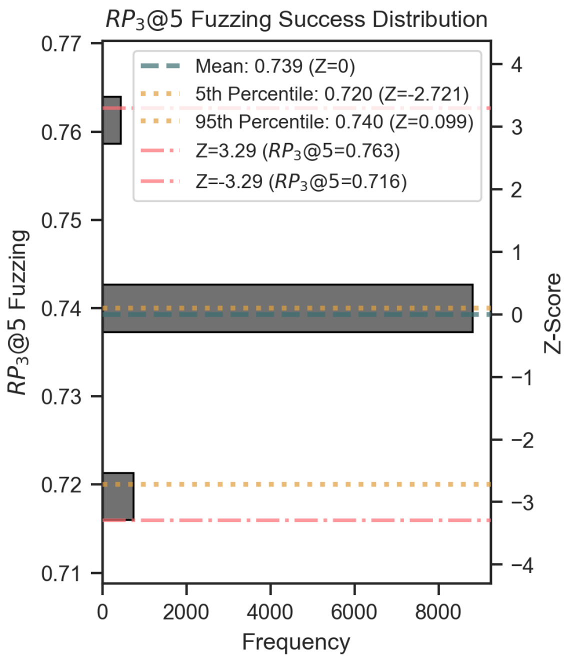


Figure 5.8: Histograms illustrating the *bootstrap-sampled* distribution of *GPT-4o-mini* $RP_3@5$ on *compilation success* (left) and *fuzzing success* (right). The x-axis indicates the frequency of $RP_3@5$ values, while the y-axis shows the actual $RP_3@5$ value. Dashed lines mark the mean (gray), the 5th and 95th percentile (yellow), as well as the *Z-Score* of 3.29. The right y-axis shows the equivalent *Z-Score*.

5.3 Summary and Relevance of the Chapter

This chapter detailed the experimental setup, including the benchmark dataset, the LLMs evaluated, as well as the utilized *deterministic* and *stochastic* perturbation strategies.

To prepare an interpretable robustness evaluation, it established the baseline performance of the *GPT-4o-mini* based code translation system and analyzed its inherent noise using the proposed methodology relying on $RP_s@k$ metrics and *Z-Scores*. This baseline provides the necessary reference point for the subsequent chapter, which will now evaluate the system's robustness when prompted with the defined perturbations.

6 Robustness Analysis under Perturbations

Building upon the experimental setup and baseline performance established in Chapter 5, this chapter evaluates the robustness of the *GPT-4o-mini* based code translation system when prompted with perturbations. It presents the results for both *deterministic* (Section 6.1) and *stochastic* (Section 6.2) perturbation strategies, as defined previously.

The analysis utilizes the baseline metrics $\boldsymbol{RP_s@k}$ and noise thresholds (*Z-Scores*) determined in Section 5.2 to differentiate between inherent model fluctuations and significant robustness deficits caused by the perturbations.

By examining the robustness of *GPT-4o-mini* under the perturbation strategies, the chapter directly proves that the proposed framework and its components can be used for a comprehensive robustness evaluation, which directly addresses RQ1 ("What methodologies and components should be integrated into a comprehensive evaluation framework to assess the robustness of an LLM-based code translation system?").

6.1 Robustness under Deterministic Perturbations

This section presents the framework's results for *GPT-4o-mini* under *deterministic perturbations*. By thoroughly analyzing the translation results according to the framework's metrics, a robustness evaluation is conducted.

6.1.1 Motivation

This experiment empirically demonstrates that the proposed methodology of the thesis answers RQ1. Specifically, this experiment utilizes the three-step framework to assess the code translation robustness of *GPT-4o-mini* under *deterministic* perturbations. Additionally, it leverages the insights of the model's baseline noise assessment to classify which perturbations cause significant robustness deficiencies.

6.1.2 Experimental Design

As detailed in previous sections, the robustness under deterministic perturbations is evaluated with $n = 5$ and $s = 1$ for 20 different perturbed datasets resulting of 12 perturbation strategies and one additional **Identity** dataset.

However, analysis revealed that the Korean **Translation** perturbation on code produced false-positive fuzzing results. The *differential fuzzer* of the code translation systems had problems identifying fuzzable functions with Korean characters. Since the fuzzer found no fuzzable functions, it mistakenly interpreted the results as successful translations. As the evaluation utilizes aggregated results, the decision was to exclude the results for this perturbation, leading to 19 perturbed datasets that were evaluated and classified by utilizing the prior baseline assessment results.

6.1.3 Results

The robustness evaluation is presented at multiple detail levels, starting with the general robustness of *GPT-4o-mini* across all deterministic perturbations.

The analysis increases its detail by distinguishing between perturbation target and perturbation levels, described in Section 4.2.2. After this, the results across the single perturbations are discussed, and lastly, the most relevant perturbations are evaluated at the file level.

As the aggregated results for target and level lose their detail, the most interesting and valuable findings are in the analysis of single perturbation strategies.

General Robustness

Table 6.1 presents the deterministic perturbation effects on *GPT-4o-mini*'s translation performance. For a direct comparison, the baseline correctness is included in the table. This data is the aggregation of all results of all deterministic perturbations. The aggregation shows that there is only a small amount of performance change.[1]

For *fuzzing success*, performance increased by 1.1%. However, a *Z-Score* of 0.518 suggests this may be due to the model's normal variability. In contrast to *fuzzing success*, *compilation success* resulted in a more outlying change. However, the *Z-Score* does not cross the threshold of 3.29 to reduce the chance of it being nondeterministic noise. Furthermore, the amount of change is 0.3% and very small. The higher *Z-Score* is a result of the model's *compilation success* baseline, which included very few fluctuations.

While the model produces robust results for the aggregation of all perturbations, it may show nondeterministic behavior for certain perturbation targets.

Correctness	Fuzzing Success		Compilation Success	
Metric	**Baseline**	**Perturbation**	**Baseline**	**Perturbation**
$\boldsymbol{RP_1}$@5 ↑	0.787	**0.796**	**1.000**	0.997
$\boldsymbol{RC_1}$@5 ↓	**0.000**	0.011	**0.000**	0.003
Z-Score	**0.000**	0.518	**0.000**	-2.44

Table 6.1: General robustness results of *GPT-4o-mini* under deterministic perturbations. The values are aggregated using the mean for **all** perturbations. The *Z-Score* shows the deviation from the baseline's $\boldsymbol{RP_1}$@5 mean.

Robustness on Deterministic Perturbation Targets

Table 6.2 presents the aggregated evaluation results across the different perturbation targets. As described in Section 4.2.2, the expectation was that *instruction* perturbations might have more impact on the model's translation performance. Comparing the *robust pass* and *robust change* values for *fuzzing success* might suggest that this expectation is confirmed. However, the low *Z-Score* values show that the change could also be due to nondeterminism, making a detailed interpretation of these results not meaningful, as it may produce completely different results in another experiment run.

[1] $\boldsymbol{RC_1}$@5 and *Z-Score* are calculated based on the original values without rounding to three decimal places. This is done for all subsequent tables.

While there is no significant change in *fuzzing success*, perturbations on comments seem to produce a significant decrease in *compilation success*, with a *Z-Score* of −4.107. However, the RC_1@5 of 0.5% is still very small, and the higher *Z-Score* is again because of the model not producing fluctuations for the baseline. Nevertheless, the data suggests that even such a small change is an outlier to the baseline performance, and subsequent evaluation details will describe if this behavior can be explained.

Metric	**Baseline**	Instruction	Comments	Code
Fuzzing Success				
RP_1@5 ↑	0.787	**0.807**	0.805	0.791
RC_1@5 ↓	**0.000**	0.025	0.023	0.005
Z-Score	**0.000**	1.138	1.041	0.214
Compilation Success				
RP_1@5 ↑	**1.000**	**1.000**	0.995	0.997
RC_1@5 ↓	**0.000**	**0.000**	0.005	0.003
Z-Score	**0.000**	0.060	-4.107	-2.504

Table 6.2: Deterministic perturbation correctness results grouped by perturbation target. The values are aggregated using the mean for the perturbations of each **target**. The *Z-Score* shows the deviation from the baseline's RP_1@5 mean.

Results per Perturbation Level

Table 6.3 details that none of the perturbation levels lead to a rigorous correctness change for *fuzzing success*.

When focusing on the more important *fuzzing success*, *instruction*, level *I*, and level *V* even produced a minor performance increase. According to RC_1@5, level *VI* produced the most change with 6%. While six percent might suggest non-robust behavior, the *Z-Score* details that these changes are still in the expected fluctuation range of the model.

However, according to the *Z-Score*, the model significantly decreases *compilation success* under level *VI* perturbations. Despite the large *Z-Score* of −16.609, the actual performance degradation is only 2%. Consequently, the model technically showed non-robust behavior in *compilation success*, but not in *fuzzing success*. So in practice, the code translation system will not be significantly affected by code with varying decision logic or expressions, as this is the focus on level *VI*.

Additionally, the meaningful changes of perturbations on comments get lost when grouped with other perturbations in level *I*. This highlights that a specific analysis of the changes by single perturbations might show more explainable findings.

Results per Perturbation

Previous aggregated analysis presented that there was no significant change in *fuzzing success* for a perturbation target or level. Yet, perturbation on comments, or those of level *VI*, suggested non-robust behavior for *compilation success*.

Figure 6.1 visualizes the translation results for the 19 perturbation strategies and the five **Identity** runs conducted in this experiment. In addition, the figure shows the baseline

Metric	Baseline	Instruction	I	II	V	VI
Fuzzing Success						
$RP_1@5\uparrow$	0.787	0.807	**0.813**	0.786	0.800	0.740
$RC_1@5\downarrow$	**0.000**	0.025	0.033	0.002	0.016	0.060
Z-Score	**0.000**	1.138	1.525	-0.080	0.750	-2.737
Compilation Success						
$RP_1@5\uparrow$	**1.000**	**1.000**	0.997	0.997	**1.000**	0.980
$RC_1@5\downarrow$	**0.000**	**0.000**	0.003	0.003	**0.000**	0.020
Z-Score	**0.000**	0.060	-2.718	-2.321	0.060	-16.609

Table 6.3: Deterministic perturbations robustness evaluation results for $s = 1$ and $k = 5$. The values are aggregated using the mean for the perturbations of each **level**. The *Z-Score* shows the deviation from the baseline's $RP_1@5$ mean.

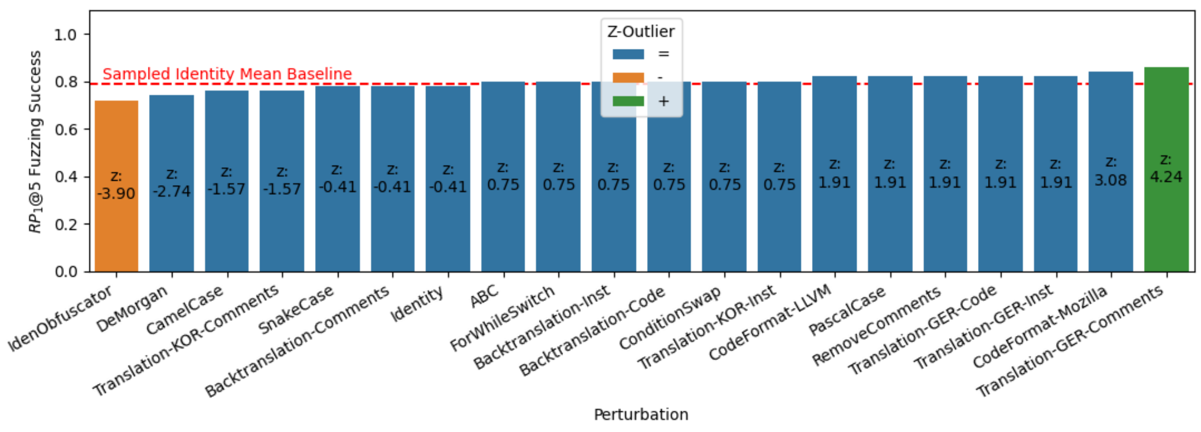
performance of the mean **Identity** across the $\binom{20}{5}$ groups, whose mean was also used for the calculation of the *Z-Score*.

Figure 6.1a adheres to the previous findings, as it shows that most perturbations did not produce a significant change to *fuzzing success*. However, by investigating the single strategies, it becomes clear that at least two perturbations produced meaningful performance changes: **IdenObfuscator** and **Translation-GER-Comments**.

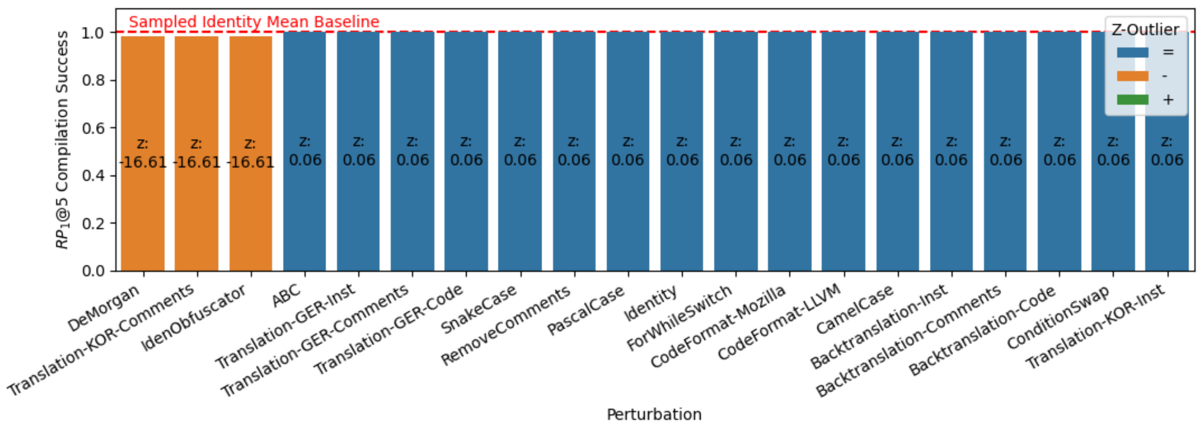
Metric	Baseline	IdenObfuscator	CodeFormat-Mozilla	Translation-GER-Comments
Fuzzing Success				
$RP_1@5\uparrow$	0.787	0.720	0.840	**0.860**
$RC_1@5\downarrow$	**0.000**	0.085	0.067	0.093
Z-Score	**0.000**	-3.899	3.075	4.238
Compilation Success				
$RP_1@5\uparrow$	**1.000**	0.980	**1.000**	**1.000**
$RC_1@5\downarrow$	**0.000**	0.020	**0.000**	**0.000**
Z-Score	**0.000**	-16.609	0.060	0.060

Table 6.4: Overview of most meaningful *deterministic* perturbations according to the *Z-Score* on *fuzzing success* for $s = 1$ and $k = 5$.

Table 6.4 details that **IdenObfuscator** yields a 8% performance degradation, resulting in a *Z-Score* of −3.899. This presents a definitive outlier to the baseline's distribution. As this perturbation obfuscates identifier names, it suggests that *GPT-4o-mini* might have difficulties with identifiers that are quite uncommon. Specifically, as this perturbation produces names, such as `_1` or `_123`, it might have a stronger effect on the embedding of these tokens than other level *II* perturbations, which is analyzed in Chapter 8. For example, **ABC** might be the most similar perturbation to **IdenObfuscator**, as it also produces identifier names without real meaning. Yet, this perturbation caused less degradation, possibly because character literals are more common as identifiers. An investigation of the error rates per perturbation in Appendix A.4.1 shows that the model tends to translate into identifiers with different names, leading to the fuzzer not being able to perform a



(a) Barplot for RP_1@5 on *fuzzing success*.



(b) Barplot for RP_1@5 on *compilation success*.

Figure 6.1: Barplots showing the particular perturbation translation results with *GPT-4o-mini* utilizing RP_1@5 for *fuzzing success* and *compilation success*. Additionally, the *Z-Score* for each perturbation is included in the bars and the color grades, whether a perturbation produced a significant change according to the 3.29 outlier rule. Orange presents a significant performance decrease, and green significant performance increase, both of which classify as non-robust behavior.

comparison of functions.

Besides **IdenObfuscator**, **Translation-GER-Comments** also produced a significant change of 9.3% performance increase. This finding is even more significant. In addition, the German translations of instructions or code identifiers also yielded a performance increase, yet they were not strong enough to be classified as outliers. However, this suggests that *GPT-4o-mini* might exhibit non-robust behavior when dealing with varying languages. When wanting to optimize the performance of the code translation system, one could think of examining other languages and finding the language that consistently works best.

Another interesting perturbation is **CodeFormat-Mozilla**. According to the *three sigma rule* [Puk94], this perturbation would have been classified as an outlier with an RC_1@5 of 0.067 and a *Z-Score* of 3.075. It seems intuitive that applying a consistent code style across all files could improve performance by unifying code structures, making them easier to read and understand. Furthermore, the model might have encountered many files with similar code style during training, ultimately improving its performance. Similarly, the **CodeFormat-LLVM** also increased the performance, but to a less significant amount. However, considering the robustness definition, a model should perform similarly on code with different formatting styles. As **CodeFormat-Mozilla** almost reached the threshold, it is considered a relevant non-robust perturbation for the subsequent discussion of the robustness of *GPT-4o-mini*. However, this consideration is with less confidence than for other perturbations.

Furthermore, the **Identity** *fuzzing success* highlights the importance of the baseline assessment. Since the **Identity** underperformed in this specific five runs of the Experiment, the $\boldsymbol{RC}_1$@5 could have been interpreted differently, or by only looking at Figure 6.1a, no perturbation would have been classified as non-robust, depending on the subjective thoughts of the human evaluator.

While *fuzzing success* is the most important correctness measurement, Figure 6.1b illustrates that **IdenObfuscator** already resulted in significantly less *compilation success*. Moreover, the perturbations **DeMorgan** and **Translation-KOR-Comments** also resulted in meaningful changes. This explains the findings in the aggregated results. Since there is only *DeMorgan* as a deterministic Level *VI* perturbation, this value could be seen in Table 6.3. Additionally, **Translation-KOR-Comments** affected the mean per comment perturbations. While the model was robust for the other three comment perturbations, it produced a significant outlier for Korean comments. However, since only **IdenObfuscator** had a real impact on the full translation correctness in *fuzzing success*, the most interesting perturbations remain **IdenObfuscator**, **Translation-GER-Comments**, and maybe **CodeFormat-Mozilla**.

Before analyzing the file-level effects for these perturbations, it should be noted why all other perturbations produced a *Z-Score* of 0.06 for *compilation success*. This can be explained by closely investigating Figure 5.7 for *compilation success*. The baseline distribution shows that there was a tiny amount of **Identity** runs, that produced an $\boldsymbol{RP}_1$@5 of 0.98, resulting in the mean not accurately being 1.0 but rather some number, very close to that, which is not visible in three decimal places. Since this result is very uncommon, most perturbations produced the more common 1.0 $\boldsymbol{RP}_1$@5, resulting in a small positive *Z-Score*.

Relevant Perturbations per File

Figure 6.2 visualizes the previously classified relevant perturbations. Recall Section 5.2.3, the interpretation of the baseline's *fuzzing success*, where results were grouped into *perfect*, *incorrect*, and *variational* files. The same separation can be used for this interpretation. Specifically, that means comparing the previous baseline's classification against the effects caused by the perturbations. The previous classification can still be identified in Figure 6.2, as the blue bar shows the performance of the *Sampled Identity*.

perfect Investigating the perfectly translated files, details that the performance increasing perturbations **CodeFormat-Mozilla** and **Translation-GER-Comments** always produced a similar $\boldsymbol{RP}_1$@5 for these files, whereas **IdenObfuscator** failed for the previously perfect files 7, 19, and 44. Analyzing the code features in Figure 5.1, this behavior might be explained by files 7, 19, and 44 having more functions or variables compared to other files classified as perfect. Consequently, the **IdenObfuscator** produces more changes to the file itself when there are more identifiers in general.

If this is actually the reason cannot be definitively proven, as the decision processes

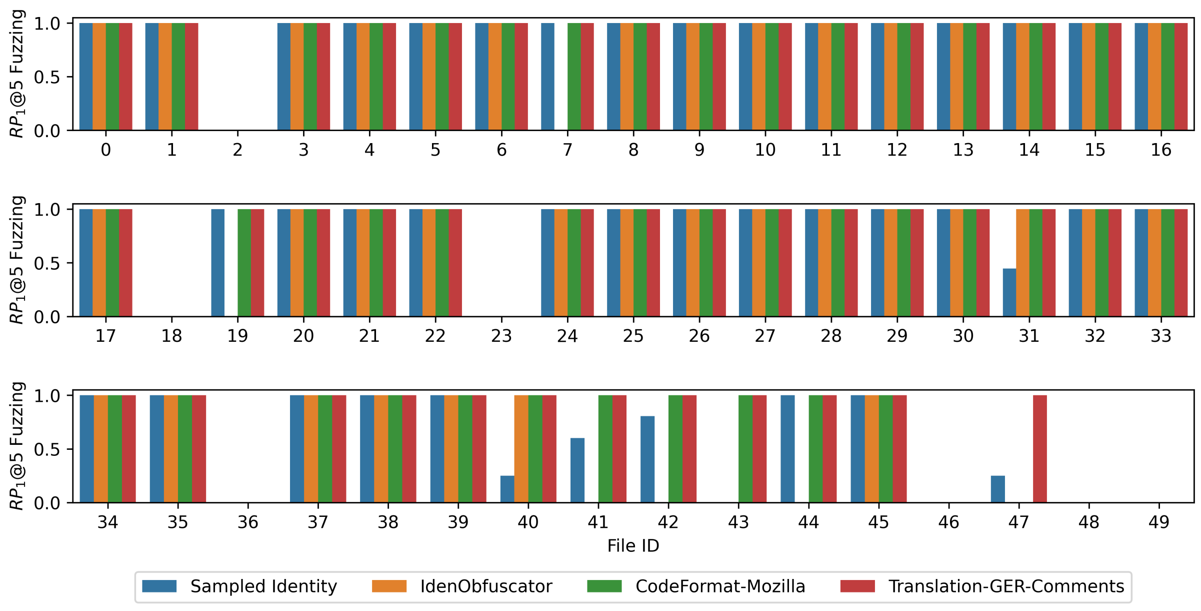


Figure 6.2: Most relevant *deterministic* perturbations per file in regard to *fuzzing success*, including the baseline *Sampled Identity*. The bar's color determines the perturbation strategy.

of LLMs are not explainable. The same applies to other hypotheses regarding why the model might fail under certain perturbations. These are guesses for the observed behavior. However, for the robustness evaluation, the primary concern is not why the model performed differently, but the fact that it did.

incorrect Most of the incorrectly translated files were also not successful for the perturbations. However, both performance-increasing perturbations were successful for file 43. The reason for this cannot be explained by code features. Specifically, poorly formatted code cannot be shown by the chosen code features, which may have suggested that the file was not accurately formatted before the perturbation, explaining why **CodeFormat-Mozilla** improved the results. Additionally, in regard to **Translation-GER-Comments**, it would have been possible that 44 is the only incorrect file containing comments, or the file with the most comments. But this is not the case, as other incorrect files have even more comments and resulted in a fuzzing failure. This suggests that translating comments may not always be enough to create performance improvements if a file cannot be accurately translated due to other unique difficulties.

variational However, investigating the variational files shows that the German translation of comments produced successful translations for all of these files. Specifically, this perturbation never yielded worse performance than the baseline, proving why the perturbation resulted in a higher $RP_1@5$ for all files. Furthermore, the variational 47 is the only file where the code format produced a worse result than the comment translation, explaining the slightly worse $RP_1@5$ for the perturbation.

Furthermore, the **IdenObfuscator** expectantly decreased the performance for most variational files, except for 31 and 40. While 40 is a file without variables, 31 involves three variables (Figure 5.1). Therefore, this cannot be simply explained by the number of identifiers and subsequent change of the perturbation. As the baseline experiment pointed out, certain errors can impact the performance of the translation results. As previously mentioned, the most impactful error rate for **IdenObfuscator** is the *fuzzing setup*, which means that the model did not produce identical identifier names, and therefore did not adhere to the actual task. However, these did not strongly affect the variational files. The actual amounts are briefly explained in Appendix A.4.1.

Considering that these were the most significant perturbations, *GPT-4o-mini* only showed minor non-robust behaviors to the files where changes would have been expected. While **IdenObfuscator** failed for some perfect files, or the **Translation** and **CodeFormat** improved a single incorrect file, the most changes were noticed on the variational files, which were prone to fluctuations even without perturbations. Consequently, the evaluation of deterministic strategies with the *feedback loops* suggests that *GPT-4o-mini* is quite robust, with tiny deficits in code formatting styles, comment language, or identifier obfuscation.

The next Experiment will show if *GPT-4o-mini* is comparably robust under *stochastic* perturbations and the less fluctuating $\boldsymbol{RP}_3@5$ metric.

6.2 Robustness under Stochastic Perturbations

This section analyzes the framework's results for *GPT-4o-mini* under *stochastic-perturbations*. As detailed previously, this experiment utilizes the concept of multiple *seeds* per perturbation. Therefore, $\boldsymbol{RP}_3@\boldsymbol{k}$ is used for the evaluation of the results. The baseline experiment highlighted that higher s in $\boldsymbol{RP}_s@\boldsymbol{k}$ prunes nondeterministic fluctuation, leading to the baseline being less variational. This may improve the interpretability of non-robust behaviors, as small performance changes display significant change to the baseline's distribution.

6.2.1 Motivation

The motivation of this experiment is to evaluate the code translation robustness of *GPT-4o-mini* under *stochastic perturbations* with the more strict metric $\boldsymbol{RP}_3@\boldsymbol{k}$. By identifying robustness deficiencies, the framework and its components provide evidence to answer RQ1.

6.2.2 Experimental Design

This experiment is conducted with 15 different perturbation datasets, each having $s = 3$ variations. Hence, the framework utilizes 45 perturbed and 3 additional **Identity** datasets. Similar to the first experiment, $n = 5$, which means that each task per $\langle perturbation, seed\rangle$ pair is translated five times. Moreover, the results are analyzed for the complete code translation setup process, including the *feedback loop* iterations.

6.2.3 Results

As for the deterministic perturbation experiment, the results are analyzed in a top-down approach. Starting with the average robustness over the aggregation for all perturbations, the evaluation gets more detailed by separating results per perturbation target or perturbation level. Lastly, the results per perturbation are examined, and the most relevant perturbations are analyzed at the file level. Consequently, the most meaningful information

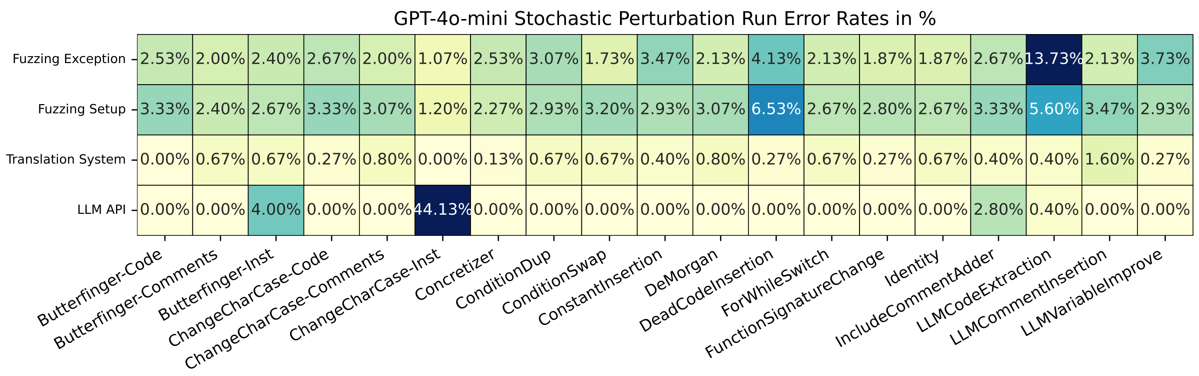


Figure 6.3: Error rates per perturbation in % for the stochastic perturbation experiment of *GPT-4o-mini*.

is in the more detailed sections. However, interpreting the general robustness capabilities among targets and levels is also part of a comprehensive evaluation.

Before starting with the top-down approach, in this experiment, it is necessary to bring up the error analysis before interpreting aggregated results. This is because Figure 6.3 highlights that **ChangeCharCase** on the instruction part of the prompt resulted in LLM API errors. As noted before, *translation system* or *LLM API* errors do not reflect robustness deficiencies, as *fuzzing setup* or *fuzzing exception* errors would do. Such a high error rate makes the robustness interpretability of **ChangeCharCase-Inst** impossible. Specifically, this high error is caused by *Azure*'s safety mechanisms, which falsely identify these perturbed instructions as *jailbreak* [Pat25] attacks, returning a *Bad Request* response. Considering that **ChangeCharCase** randomly switches a character's case with quite high probability (i.e., 0.30), it seems reasonable that the system may classify such a prompt as *jailbreak*. Knowing this, it is necessary to exclude **ChangeCharCase-Inst** from the aggregated robustness analysis.

Moreover, the same behavior can be seen for **Butterfinger-Inst**. However, the error rate is much smaller, leaving room for a robustness evaluation. In addition, **IncludeCommentAdder** also shows noticeable *LLM API* errors. However, these stem from partial *connection errors* in a single run. While this slightly reduces the chance of choosing a correct run in k runs, it should not have a tremendous effect. So, both of these perturbations are included in the aggregation, yet this information is important to keep in mind for the interpretation.

General Robustness

Table 6.5 details the aggregated translation results for the stochastic perturbations. The table shows a performance degradation of 3.2% under stochastic perturbations. Despite this being a relatively moderate decrease, the *Z-Score* of -3.28 almost suggests significant non-robust behavior. This matched the expectation for $RP_3@k$, as it resulted in a smaller standard deviation in the baseline distribution, thus reducing susceptibility to fluctuations. Subsequently, minor percentage changes already present outliers to the baseline performance.

It might seem surprising that *compilation success* has a lower *Z-Score* than *fuzzing success*, as this was the opposite behavior for deterministic perturbations. However, recalling the baseline on $RP_3@5$'s *compilation success* showed that this involved more fluctuations

Correctness Metric	Fuzzing Success Baseline	Fuzzing Success Perturbation	Compilation Success Baseline	Compilation Success Perturbation
$\boldsymbol{RP}_3$@5 ↑	0.739	**0.716**	**0.993**	0.972
$\boldsymbol{RC}_3$@5 ↓	**0.000**	0.032	**0.000**	0.021
Z-Score	**0.000**	-3.28	**0.000**	-2.226

Table 6.5: General robustness results of *GPT-4o-mini* under *stochastic* perturbations. The values are aggregated using the mean for **all** working perturbations. The *Z-Score* shows the deviation from the baseline's $\boldsymbol{RP}_3$@5 mean.

than for $s = 1$. As *compilation success* is almost perfect with $\boldsymbol{RP}_3$@5 0.993, slight variations among the $s = 3$ samples increase the standard deviation in this case. Thus, *stochastic perturbations* cause larger performance changes compared to *deterministic* ones, but these changes are not statistically significant when aggregated.

The following analysis among the different levels of detail will investigate, under which conditions *GPT-4o-mini* showed the most robustness deficiencies.

Robustness on Stochastic Perturbation Targets

Table 6.6 shows the aggregated translation results for the three different perturbation targets. This explains where the most robustness issues stem from. Specifically, perturbations on instructions produce an 8% performance loss in regard to *fuzzing success*, resulting in a *Z-Score* of -8.36. This accommodates the initially expected behavior that perturbations on instruction might have the strongest influence on perturbation results. Additionally, these perturbations led to a substantial decrease in *compilation success* by 11.4%. The *stochastic* perturbations contain the perturbations that produce noise with typos, where deterministic perturbations are used rather than intentional paraphrasing approaches on the instructions. So it is quite reasonable that the *stochastic* perturbations adhered more to the expectation, as it is more difficult to maintain the intent in noisy instructions. Chapter 8 will investigate if such difficulties can be predicted by the semantic similarity analysis.

Besides instruction perturbations, code perturbations almost resulted in *Z-Score* significant performance degradation, with a *Z-Score* of -3.003. Considering that the $\boldsymbol{RP}_3$@5 of 0.718 represents the mean, there have to be certain perturbations, or even levels, that resulted in a *Z-Score* significant robustness deficiency.

Results per Perturbation Level

Table 6.7 presents that *GPT-4o-mini* showed significant robustness deficits only for level *IV*, except at the instruction level, which already has been examined. Level *IV* perturbations lead to a performance loss of 9.4%, which is very strong considering level *IV* aggregates two perturbation strategies. This also shows in a clear outlying *Z-Score* of -9.77. The next section examines whether this result was caused by one specific perturbation or if both perturbations significantly impacted *GPT-4o-mini*.

Besides that, *compilation success* measured no significant changes, except for the instruction perturbations.

Metric	**Baseline**	Instruction	Comments	Code
Fuzzing Success				
$\boldsymbol{RP}_3@5 \uparrow$	**0.739**	0.680	0.733	0.718
$\boldsymbol{RC}_3@5 \downarrow$	**0.000**	0.080	0.008	0.029
Z-Score	**0.000**	-8.360	-0.841	-3.003
Compilation Success				
$\boldsymbol{RP}_3@5 \uparrow$	**0.993**	0.880	0.987	0.986
$\boldsymbol{RC}_3@5 \downarrow$	**0.000**	0.114	0.007	0.007
Z-Score	**0.000**	-11.911	-0.683	-0.753

Table 6.6: *Stochastic* perturbation correctness results grouped by perturbation target. The values are aggregated using the mean for the perturbations of each target. The *Z-Score* shows the deviation from the baseline's $\boldsymbol{RP}_3@5$ mean.

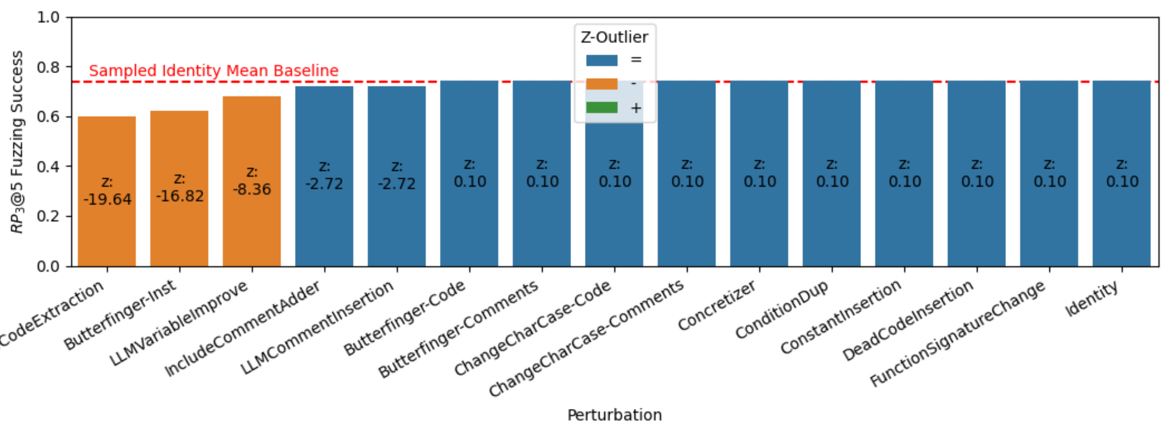
Metric	**Baseline**	Instruction	I	II	III	IV	VI
Fuzzing Success							
$\boldsymbol{RP}_3@5 \uparrow$	0.739	0.680	0.733	0.720	0.733	0.670	**0.740**
$\boldsymbol{RC}_3@5 \downarrow$	**0.000**	0.080	0.008	0.026	0.008	0.094	0.001
Z-Score	**0.000**	-8.360	-0.841	-2.721	-0.841	-9.770	0.099
Compilation Success							
$\boldsymbol{RP}_3@5 \uparrow$	0.993	0.880	0.987	0.980	0.987	0.980	**1.000**
$\boldsymbol{RC}_3@5 \downarrow$	**0.000**	0.114	0.007	0.013	0.007	0.013	0.007
Z-Score	**0.000**	-11.911	-0.683	-1.384	-0.683	-1.384	0.721

Table 6.7: *Stochastic* perturbations robustness evaluation results for $s = 3$ and $k = 5$. The values are aggregated using the mean for the perturbations of each level. The *Z-Score* shows the deviation from the baseline's $\boldsymbol{RP}_3@5$ mean.

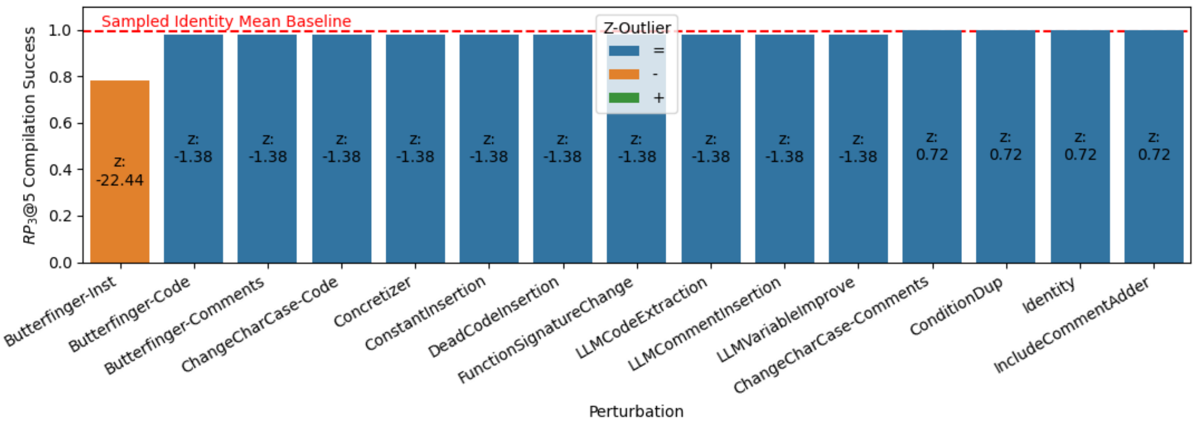
Results per Perturbation

Figure 6.4 highlights that most of the stochastic perturbations did not cause significant performance changes. However, there are three significant performance degradations, each originating from perturbations at different levels.

Among these significant perturbations, **LLMCodeExtraction** at level *IV* caused the largest drop in *fuzzing success*, which explains the *outlying Z-Score* observed at level *IV*. Specifically, under this perturbation, *GPT-4o-mini* showed a 18.8% decrease in *fuzzing success*, even though the *compilation success* was not strongly affected (see Table 6.8). According to Figure 6.3, **LLMCodeExtraction** produced a *fuzzing exception* error rate of 13.73%, and a *fuzzing setup* error rate of 5.6%. Consequently, this perturbation caused an unusually high number of errors, ultimately affecting *fuzzing success*. At first glance, the involvement of an LLM in the perturbation might suggest that the code extraction produced erroneous *C* code. However, the *syntax-check* implemented in *Step I* of the framework proved that there were no syntax errors introduced by the perturbation itself. Furthermore, *GPT-4o-mini* achieved near-baseline performance regarding *compilation success* for the translated *Rust* code. Thus, *GPT-4o-mini* must have introduced inconsistencies between



(a) Barplot for $RP_3@5$ on *fuzzing success*.



(b) Barplot for $RP_3@5$ on *compilation success*.

Figure 6.4: Barplots showing the particular perturbation translation results with *GPT-4o-mini* utilizing $RP_3@5$ for *fuzzing success* and *compilation success*. Additionally, the *Z-Score* for each perturbation is included in the bars and the color grades, whether a perturbation produced a significant change according to the 3.29 outlier rule. Orange presents a significant performance decrease, and green significant performance increase, both of which classify as non-robust behavior.

the *C* and *Rust* code, likely resulting from the increased complexity due to encapsulated function calls. Specifically, the perturbation aims to refactor and extract certain code snippets into functions, which are then invoked. This may add more complexity to the control flow, leaving more room for errors in the translation. This assumption is backed up by a small analysis of the **Identity** and **LLMCodeExtraction** datasets.

The **Identity** code contained a total of 18 nested function calls across all 50 files, whereas the **LLMCodeExtraction** perturbation contained a notably higher average of 52.33 nested function calls across the 50 perturbed files per *seed*. This increase in nested calls indicates higher complexity, which may cause translation issues.

This is an interesting robustness evaluation result, as it highlights significant weaknesses of *GPT-4o-mini* when translating code with varying complexity due to nested calls. To further prove this hypothesis, future work could evaluate the robustness on a different benchmark dataset containing inherently more complex *C* files.

Besides **LLMCodeExtraction** causing robustness issues, **Butterfinger** on instructions

Metric	**Baseline**	LLMCodeExtraction	Butterfinger-Inst	LLMVariableImprove
Fuzzing Success				
$RP_3@5 \uparrow$	**0.739**	0.600	0.620	0.680
$RC_3@5 \downarrow$	**0.000**	0.188	0.161	0.080
Z-Score	**0.000**	-19.639	-16.820	-8.360
Compilation Success				
$RP_3@5 \uparrow$	**0.993**	0.980	0.780	0.980
$RC_3@5 \downarrow$	**0.000**	0.013	0.215	0.013
Z-Score	**0.000**	-1.384	-22.438	-1.384

Table 6.8: Overview of most meaningful *stochastic* perturbations according to the *Z-Score* on *fuzzing success* for $s = 3$ and $k = 5$.

resulted in significant performance loss. However, this perturbation already had a drastic impact on the *compilation success*. That shows that the **Butterfinger** perturbation on the instruction resulted in a higher difficulty for the model in understanding the general task. While some failures may stem from the 4% error rate because of false *jailbreak* filters, that does not justify the drastic performance loss of 21.5% regarding *compilation success*. Recalling the deterministic perturbations, this is the only perturbation that drastically reduced the *compilation success* to an $RP_s@5$ of 0.78 (Table 6.8). Considering that only compilable code can be tested for *fuzzing success*, a robust change of 16% is understandable and suggests that predominantly the baseline's *incorrect* files were not translated into compilable code. Before investigating this hypothesis, the **LLMVariableImprove** has to be analyzed, as it also caused a significant robust change.

As Figure 6.3 shows, **LLMVariableImprove** shows slight elevation for *fuzzing exception* and *fuzzing setup* errors. However, other perturbations have similar error rates and did not result in a significant robustness deficiency. Subsequently, this perturbation has to reveal other problems, which may show in a more detailed analysis per file.

Relevant Perturbations per File

Figure 6.5 illustrates the relevant perturbations *fuzzing success* per file. Similar to Section 5.2.3, the grouping of *perfect*, *incorrect*, and *variational* files is used to improve interpretation. However, as Figure 6.5 shows, there is not a single perturbation that improved an *incorrect* file. Furthermore, the baseline produced only two *variational* files with $s = 3$, where one of which (42) has never been translated successfully under perturbation and the other one (44) was deemed successful for all perturbations. Subsequently, the analysis only focuses on the baseline's *perfect files*.

Starting with **LLMCodeExtraction**, *GPT-4o-mini* fails for seven of the *perfect* files. However, it could be identified no reason for the failure, because files with lower token counts and also files with higher token counts fail. In addition, there is no code feature that stands out, and failure may just be caused by the fact that the difficulty for each file is increased because of the introduced encapsulation. With $s = 3$, it is even harder for the model to create a successful translation.

Same as **LLMCodeExtraction**, the **Butterfinger-Inst** perturbation fails without an obvious pattern at first glance. However, upon closer inspection, it became clear that some

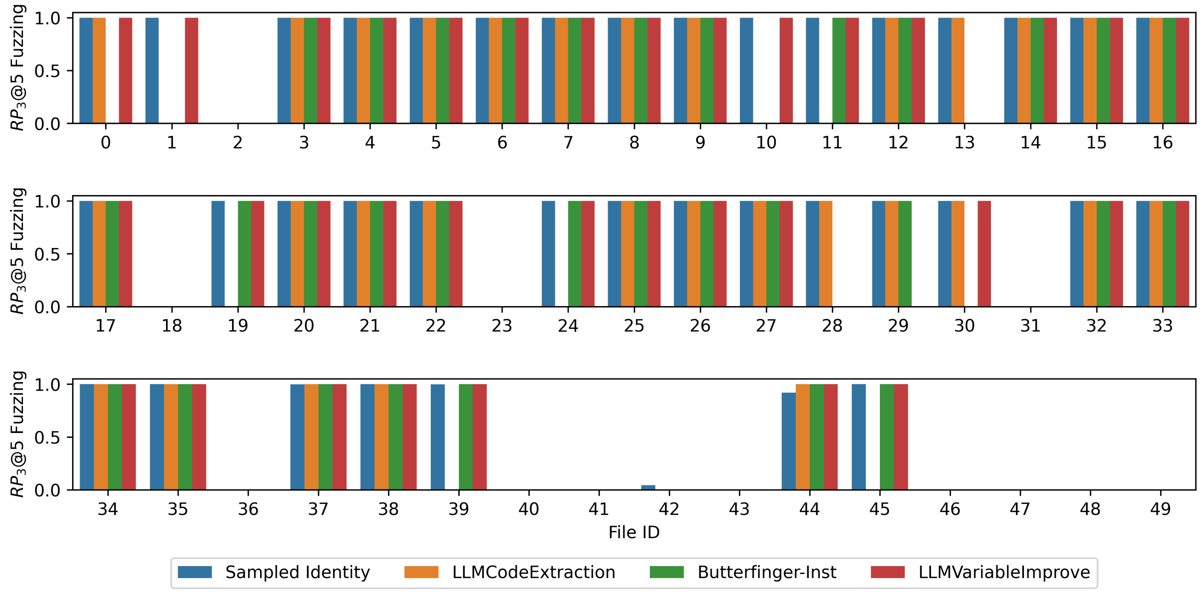


Figure 6.5: Most relevant *stochastic* perturbations per file in regard to *fuzzing success*, including the baseline *Sampled Identity*. The bar's color determines the perturbation strategy.

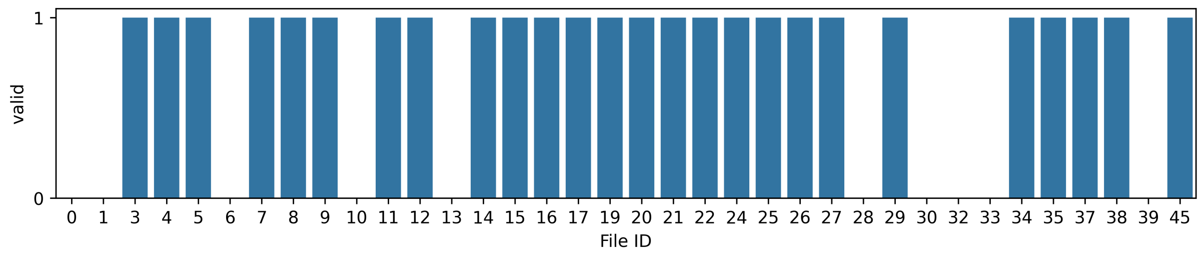


Figure 6.6: Instruction validity after **Butterfinger-Inst** for the baseline's perfect *fuzzing success* files. The plot also contains the closely perfect files 37 and 39 that are not exactly at RP_3@5 of 1.0. A file is valid if and only if all three variations contain the correct spelling of *Rust*.

perturbed instructions modified the term "*Rust*", corresponding exactly to the scenario described in Section 4.2.2. To quantify this effect, instructions were explicitly checked for correct spelling of *Rust*. A file is considered *valid* only if all instructions across all seeds correctly mention *Rust*, otherwise, it is labeled *invalid*. Figure 6.6 shows this validity assessment for the baseline's *perfect* files.

Using this categorization, the *Pearson correlation coefficient* [SBS18] between instruction validity and *fuzzing success* was calculated. The coefficient, measuring the linear relationship between validity and performance, yields a value of 0.721, indicating a *strong correlation* according to [SBS18]. Consequently, *GPT-4o-mini* exhibited non-robust behavior primarily when instructions were unclear or misleading. However, this is considered a reasonable outcome and should not negatively affect the general robustness assessment of the model.

Lastly, **LLMVariableImprove** caused translation failures for files 13, 28, and 29. File 13 is an expected failure, as the *syntax check* in *Step I* noticed that the perturbation introduced a syntax error for this file, making *differential fuzzing* impossible. Specifically, the perturbation modified the identifier `int r1` to just `int 1`, which is not allowed. Files

28 and 29 also resulted in errors, namely *Fuzzing Exception* and *Fuzzing Setup*. In one of which the perturbation introduced almost duplicate identifier names, e.g. `my_identifier` and `My_identifier`, one for a variable and one for a function. While this was not a problem in *C*, it resulted in an erroneous *Rust* translation, where the same identifiers for a variable and function are not allowed, regardless of the casing strategy. Considering that **LLMVariableImprove** only failed in three *perfect* files, the perturbation should no longer be considered as a significant non-robust perturbation.

Consequently, *GPT-4o-mini* only shows meaningful non-robust behavior for **LLM-CodeExtraction**. Besides this perturbation, the model proved to be robust, especially considering that $s = 3$ presents a higher difficulty, as the model has to provide successful translations for all seeds in a run.

6.3 Summary of Robustness Findings

Employing the robustness evaluation framework detailed in Chapter 4 and using the baseline from Section 5.2, the analysis of *GPT-4o-mini*'s behavior under perturbations suggests the model is generally robust, exhibiting only minor deficits for specific strategies. Combining the behavior under both *deterministic perturbations* and *stochastic perturbations*, neither a single perturbation target nor a specific perturbation level consistently caused significant non-robustness. While for the *stochastic perturbation* there was the impression that perturbations on instruction or on level *IV* produce significant robustness deficits, deeper investigation showed that these outliers were always caused by single strategies, which drastically affected the aggregated performance of a classification.

Focusing on the *Z-Score* outlier strategy, the experiments revealed that only five perturbations in total resulted in an absolute *Z-Score* greater than 3.29 for the most relevant correctness measure, *fuzzing success*. File-level examinations showed that **Butterfinger-Inst** as well as **LLMVariableImprove** should not be considered in the assessment. As the former produced instructions with strongly modified intent and the second one only being a close outlier, where some failures originated from not fuzzable or not translatable *C* code, making a successful translation impossible. Furthermore, file-level analysis suggested that **CodeFormat-Mozilla** can be considered a perturbation producing significant non-robust behavior. However, with a *Z-Score* of 3.075, the confidence is slightly lower.

This leaves four relevant perturbations: **Translation-GER-Comments**, **CodeFormat-Mozilla**, **IdenObfuscator**, and **LLMCodeExtraction**.

Interestingly, translating comments into German significantly improved translation performance, which unexpectedly indicated non-robust behavior due to the sensitivity to comment language. This is an interesting phenomenon, and whether this is because of the benchmark dataset containing major amounts of English code snippets from a German company, where primarily native German speakers write English code, or if it is a general characteristic of *GPT-4o-mini* cannot be judged at this point. Nonetheless, it presents a robustness deficit and could be investigated in other works.

Additionally, **CodeFormat-Mozilla** also increased the translation performance of the code translation system. The reason for that could not be definitely found, but it suggests that the model might perform better when code follows a consistent style, especially the style guide of *Mozilla*.

Besides that, the **IdenObfuscator** decreased the likelihood of *GPT-4o-mini* producing successful translations. That may stem from the perturbation introducing uncommon identifier names, where *GPT-4o-mini* did not manage to produce identical function names, resulting in a fuzzing failure. However, this perturbation is probably the strategy with the least *real-world relevance*, making robustness deficits for this strategy less dramatic.

Lastly, the perturbation that the model showed the strongest robustness deficits is **LLMCodeExtraction**. Upon close investigation, there was no definitive factor identifiable, why the model failed more often under this perturbation. The only noteworthy finding was that this perturbation increased the number of nested functions, suggesting that the model may have difficulties handling more complex control flows. This impression could be investigated by collecting a separate benchmark dataset that specifically involves code files with high amounts of nested functions.

When excluding the two problematic perturbations **Butterfinger-Inst** and **LLM-VariableImprove**, in total, the code translation system with *GPT-4o-mini* only showed robustness deficits in three of 31 strategies. The fact that the framework could be used to discover these findings underlines that its components are valuable for a robustness evaluation, which is important when recalling RQ1. However, since the evaluation was conducted with the *feedback loops* in *Step II*, it is not yet clear whether the overall robust behavior stems from *GPT-4o-mini*'s characteristic, or if the auto repairing *feedback loop* approach takes an influence on the model's robustness. Consequently, the upcoming chapter investigates the influence of *feedback loops* on the robustness.

7 Assessing the Impact of Feedback Loops on Robustness

The previous chapter discussed the robustness evaluation results of the default code translation system with *GPT-4o-mini*. Recall that the code translation system incorporates multiple auto-repairing *feedback loops* that may not only have improved the performance of the model but also affected its robustness. Consequently, this chapter extends that analysis by specifically investigating whether and how incorporating *feedback loops* affected the robustness, directly addressing RQ3: "Does incorporating a *feedback loop* strategy impact robustness?"

In this chapter, the thesis systematically evaluates how *feedback loops* influence robustness against *deterministic* and *stochastic perturbations*. Initially, it analyzes the baseline performance to establish a reference point across the iterations, followed by detailed examinations of perturbations. The results provide clear insights into the strengths and limitations of the *feedback loop* mechanism in maintaining translation robustness.

The entire examination can be conducted on the data of the previous experiments, as the unique result per feedback iteration is stored. Therefore, this chapter uses the same experimental setup and also compares the robustness assessment for certain strategies with and without *feedback loops*.

7.1 Motivation

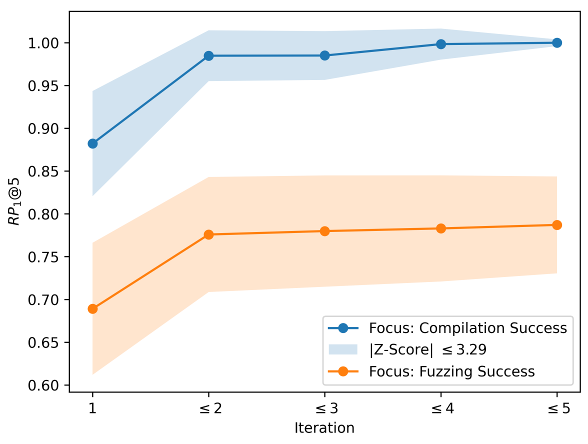
The integration of *feedback loops* within the code translation system aims to improve translation performance by iteratively refining failing outputs. However, while these *feedback loops* may enhance translation success, they could also influence the model's robustness. In theory, non-robust behavior could be mitigated by employing iterative *feedback loops* that guide the model towards more stable outputs. To get a profound understanding of the model's robustness, it is essential to analyze the influence of *feedback loops*, which is the motivation behind RQ3. Therefore, this chapter aims to investigate the impact of *feedback loops* on the model's inherent robustness characteristics by analyzing the translation performance across different iteration steps. These insights are finally used to answer RQ3 for *GPT-4o-mini*.

7.2 Feedback Loops on the Baseline Performance

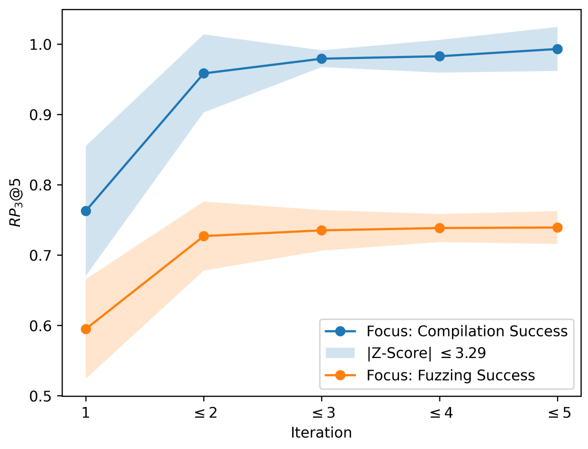
Before analyzing the *feedback loops* influence on the robustness of the model, it is necessary to present their influence on the baseline performance, with only **Identity**.

Figure 7.1 visualizes the influence of the *feedback loops* on the general baseline performance, also showing the amount of nondeterministic variations across iterations.

Without *feedback loops*, the system clearly has a smaller chance of producing correct translations, even only considering *compilation success*, where the system produced reliably



(a) Baseline performance for $s = 1$.



(b) Baseline performance for $s = 3$.

Figure 7.1: Lineplot showing the baseline performance of *GPT-4o-mini* across the different iterations of the *feedback loops*. The fill around the lines presents the nondeterministic noise area with absolute *Z-Scores* ≤ 3.29.

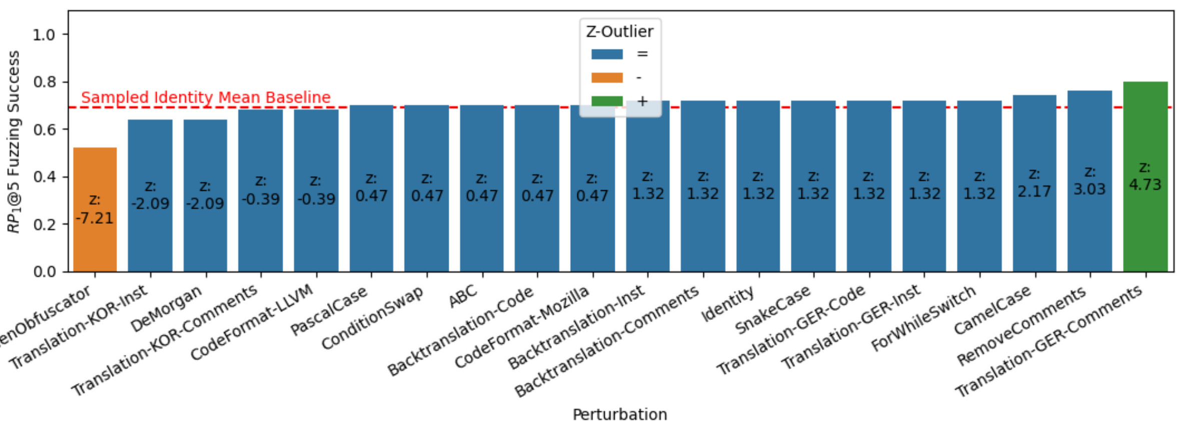
good results with the *feedback loops* in Chapter 5.

For both $\boldsymbol{RP}_1@5$ and $\boldsymbol{RP}_3@5$, the strongest performance improvement is achieved after the first feedback iteration (≤ 2). Further iterations only slightly improved *fuzzing success* and *compilation success*, which goes in hand with the previous findings described in Section 4.3.4. Specifically, the decision to use a *max_retries* of five was made because prior testing showed that after some amount of iterations, the model is not able to fix the errors, leading to unnecessarily many LLM calls. The plot details that when wanting to save on LLM calls, it could be a good trade-off to choose a smaller *max_retries*.

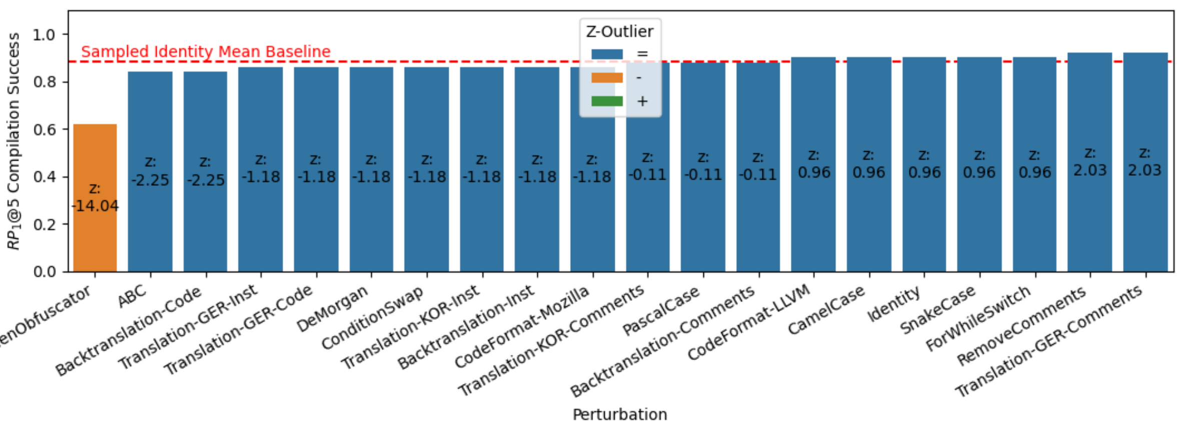
However, the plot also demonstrates that the amount of fluctuations can be reduced by applying more iterations. While this only slightly shows for *fuzzing success* on $\boldsymbol{RP}_1@5$, the other correctness measures present significantly smaller areas around the mean in higher iterations. Therefore, *max_retries* of five is beneficial for the robustness evaluation, as in early iterations it is harder to distinguish between performance changes arising from nondeterministic noise or changes caused by perturbations.

An additional observation is that for $\boldsymbol{RP}_3@5$, there appear to be groups in iteration ≤ 2 that yield higher *fuzzing success* than groups in iteration ≤ 5. In reality, however, this cannot occur. Recall that the 3.29 *Z-Score* threshold does not represent actual performance values, but rather a range defined by scaling the standard deviation. Once an iteration achieves *fuzzing success*, the *feedback loop* terminates, making it impossible to obtain a worse result in later iterations. However, the situation differs for *compilation success*. A successful compilation may still fail during fuzzing, thus triggering another iteration, which could theoretically lead to compilation failure.

The next section focuses on *deterministic* and *stochastic* perturbations to investigate how feedback iterations influence the translation robustness, thus addressing RQ3 more directly.



(a) Barplot for RP_1@5 on *fuzzing success* in iteration one.



(b) Barplot for RP_1@5 on *compilation success* in iteration one.

Figure 7.2: Barplots showing the particular *deterministic perturbation* translation results with *GPT-4o-mini* in *iteration one* utilizing RP_1@5 for *fuzzing success* and *compilation success*. Additionally, the *Z-Score* for each perturbation is included in the bars and the color grades, whether a perturbation produced a significant change according to the 3.29 outlier rule. Orange presents a significant performance decrease, and green significant performance increase, both of which classify as non-robust behavior.

7.3 Deterministic Perturbations

To evaluate the impact of *feedback loops* on robustness, it is not necessary to aggregate results for different perturbation targets or perturbation levels. The investigation first shows the impact on single results per perturbation and later concludes the impact on robustness.

Figure 7.2 visualizes that in iteration one, there are almost the same perturbations causing significant deviations as with five iterations. Recall that **IdenObfuscator** resulted in a RC_1@5 of 0.085 with *feedback loops* (see Table 6.4 and Figure 6.1). Without them, **IdenObfuscator** resulted in an even more drastic correctness loss, with an RC_1@5 of 0.245 (see Table 7.1). Comparing Figure 7.2b and Figure 6.1b highlights that the *feedback loops* managed to elevate *compilation success* for this perturbation, to only 2% less RP_1@5 than the baseline. Furthermore, the comparison shows that the other *compilation success* significant perturbations in iteration five, namely **DeMorgan** and **Translation-KOR-**

Metric	**Baseline**	IdenObfuscator	Translation-GER-Comments
Fuzzing Success			
$RP_1@5 \uparrow$	0.689	0.520	**0.800**
$RC_1@5 \downarrow$	**0.000**	0.245	0.161
Z-Score	**0.000**	-7.211	4.733
Compilation Success			
$RP_1@5 \uparrow$	0.882	0.620	**0.920**
$RC_1@5 \downarrow$	**0.000**	0.297	0.043
Z-Score	**0.000**	-14.038	2.031

Table 7.1: Overview of the most meaningful *deterministic* perturbations in *iteration one* according to the *Z-Score* on *fuzzing success* $s = 1$ and $k = 5$.

Comments, are not significantly different in iteration one. However, the actual amount of change is higher in iteration one, but since the model also produced more fluctuations in the first iteration, they were not yet significant outliers with the *Z-Score* threshold.

Translation-GER-Comments apparently resulted in a quite high $RP_1@5$ in iteration one (i.e., 0.8). This is not only 16.1% better than the *Sampled Identity* baseline in iteration one, but presents a higher *fuzzing success* than the baseline after five iterations.

Recall that the prior chapter considered **CodeFormat-Mozilla** as non-robust behavior causing perturbation with *feedback loops*. Without *feedback loops*, this behavior could not be seen. However, **CodeFormat-Mozilla** also did not cross the 3.29 *Z-Score* threshold with *feedback loops* and was classified non-robust with less confidence (see Section 6.1.3).

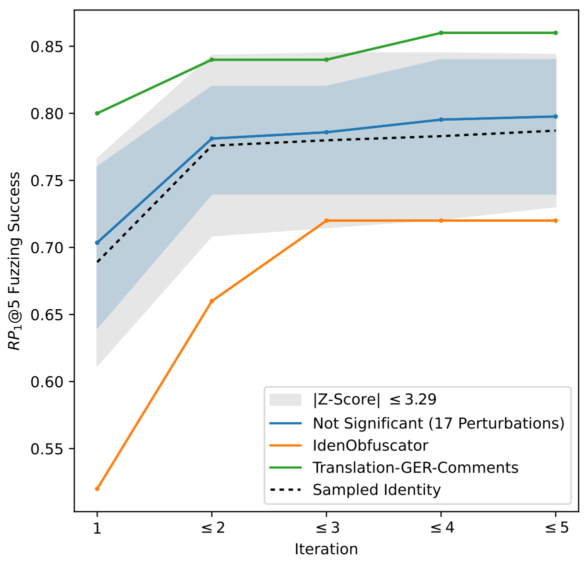
Overall, translation success is generally worse without *feedback loops*. However, for *deterministic perturbations*, the number of non-robust perturbations appears to be similar with and without *feedback loops*.

The impact of feedback loops becomes more evident in Figure 7.3, where perturbations are grouped based on whether they exceed the 3.29 *Z-Score* threshold during iteration one. Specifically, those that were not significant are represented with the blue area and the blue line describing their mean.

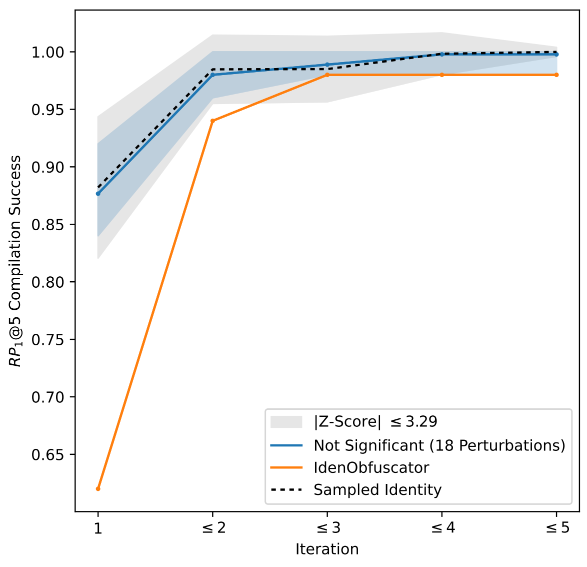
For fuzzing success in Figure 7.3a, the outlying perturbations initially move closer to the *Z-Score* region during the first two iterations. Beyond this point, **IdenObfuscator** remains unchanged while **Translation-GER-Comments** continues to improve, clearly distinguishing both from the baseline. Therefore, the *feedback loops* reduce the extreme variations of these perturbations, yet they do not make them fully robust even after five iterations.

A similar trend is observed for *compilation success* in Figure 7.3b. Although a substantial gap exists in iteration one, by iteration three **IdenObfuscator** closely aligns with the baseline and then stabilizes without further improvement. In addition, some perturbations that were initially non-significant do not improve along with the baseline and eventually fall outside the noise range by iteration five.

Overall, these findings suggest that *feedback loops* enhance robustness by narrowing performance variance. However, as the baseline fluctuations decrease with successive iterations, the loops also expose perturbations that deviate from the baseline. Thus, even though the total number of non-robust perturbations remains unchanged, the magnitude of deviations is reduced, which is a result that is particularly significant in practical



(a) RP_1@5 for *fuzzing success* across the *feedback loop* iterations.



(b) RP_1@5 for *compilation success* across the *feedback loop* iterations.

Figure 7.3: Lineplot showing RP_1@5 values for the different iterations under *deterministic perturbations*. The perturbations get *Z-Score* classified in *iteration one*, and the two perturbations that were significant outliers in iteration one are represented with a unique line. In addition, the dashed line shows the sampled **Identity** baseline and the area of nondeterministic noise, according to the absolute *Z-Score* being ≤ 3.29.

applications.

The *stochastic perturbations* will show if similar patterns occur and if *feedback loops* may reduce deviations while not reducing the number of non-robust perturbations.

7.4 Stochastic Perturbations

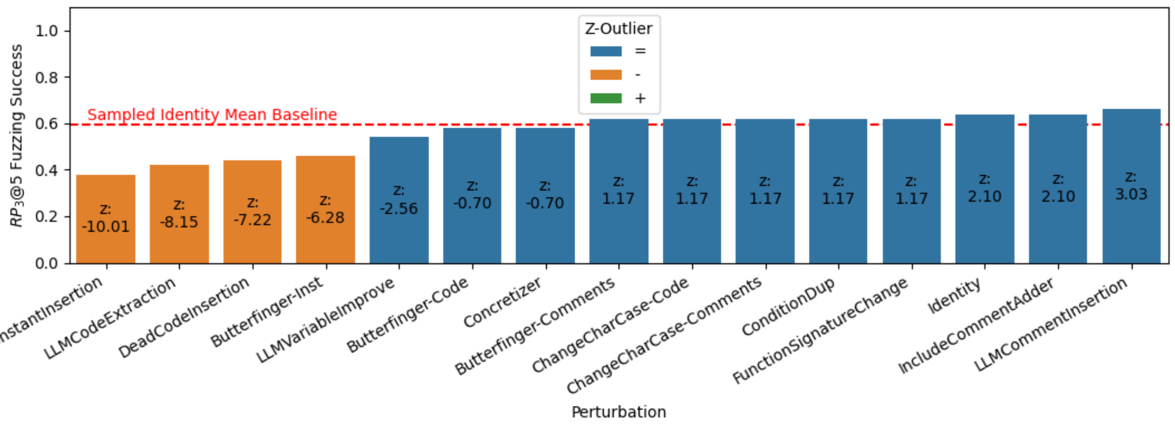
To evaluate the impact of *feedback loops* under the *stochastic perturbations*, this section also utilizes the robustness information under the single perturbations at different iterations.

Recall that with at most five iterations, the model showed non-robust behavior for three strategies, namely **LLMCodeExtraction**, **Butterfinger-Inst**, and **LLMVariableImprove**. After a detailed investigation, **Butterfinger-Inst** was excluded because many perturbations modified the intent of the instruction too drastically. In addition, **LLMVariableImprove** was excluded because some failures were inevitably caused by the perturbation, making the amount of performance change not significant enough.

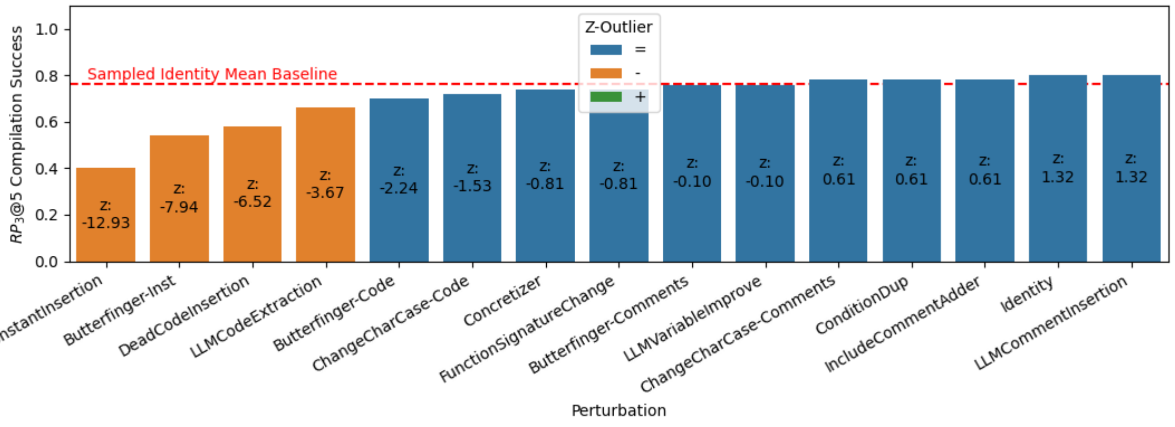
Figure 7.4 displays that there are four significantly outlying perturbations at *iteration one*. While **LLMCodeExtraction** and **Butterfinger-Inst** are among these, **LLMVariableImprove** did not result in enough change to be significant.

Besides these perturbations *GPT-4o-mini* underperformed for **ConstantInsertion** and **DeadCodeInsertion**. Interestingly, these are similar perturbations, with **ConstantInsertion** adding random constants and **DeadCodeInsertion** adding more profound dead-code snippets.

ConstantInsertion already produces drastic loss of 47.6% for *compilation success* (see Table 7.2). Further investigation showed that this actually is caused by the *Rust* code



(a) Barplot for RP_1@5 on *fuzzing success* in iteration one.



(b) Barplot for RP_1@5 on *compilation success* in iteration one.

Figure 7.4: Barplots showing the particular *stochastic perturbation* translation results with *GPT-4o-mini* in *iteration one* utilizing RP_1@5 for *fuzzing success* and *compilation success*. Additionally, the *Z-Score* for each perturbation is included in the bars and the color grades, whether a perturbation produced a significant change according to the 3.29 outlier rule. Orange presents a significant performance decrease, and green significant performance increase, both of which classify as non-robust behavior.

failing the linting check with *clippy*. Recall that the *compilation success* is the combination of the translation succeeding a compilation and a linting check. As both success values are mostly similar, the decision was to merge them to simplify the interpretation. However, for this example, it is beneficial to distinguish between those measures. The reason for that can be seen in Figure 7.5, where the *compiler success* is almost as comparable large as for the *Sampled Identity*[1]. However, *clippy success* is significantly smaller, also resulting in a smaller *fuzzing success*. The reason for this behavior is that the **ConstantInsertion** regularly introduces a constant `PI:=3.14159`. *Clippy* complains about that and wants the programmer to use *Rust*'s integrated π constant. This is an easy fix for the model, which explains why the perturbation produces no significant changes in later iterations.

For **DeadCodeInsertion** it is different. Under this perturbation, the model produces noticeably less *compiler success*. However, a clear reason for these failures could not be

[1]Since *clippy success* is identical to *compiler success*, there is no blue bar for the *Sampled Identity*.

Metric	**Baseline**	ConstantInsertion	DeadCodeInsertion	LLMCodeExtraction
Fuzzing Success				
$RP_3@5\uparrow$	**0.595**	0.380	0.440	0.420
$RC_3@5\downarrow$	**0.000**	0.361	0.260	0.294
Z-Score	**0.000**	-10.009	-7.215	-8.147
Compilation Success				
$RP_3@5\uparrow$	**0.763**	0.400	0.580	0.660
$RC_3@5\downarrow$	**0.000**	0.476	0.240	0.135
Z-Score	**0.000**	-12.932	-6.517	-3.666

Table 7.2: Overview of most meaningful *stochastic* perturbations in *iteration one* according to the *Z-Score* on *fuzzing success* and *compilation success* $s = 3$ and $k = 5$.

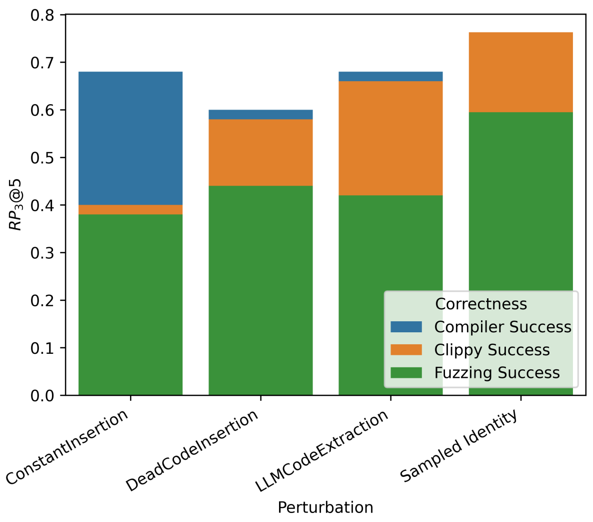
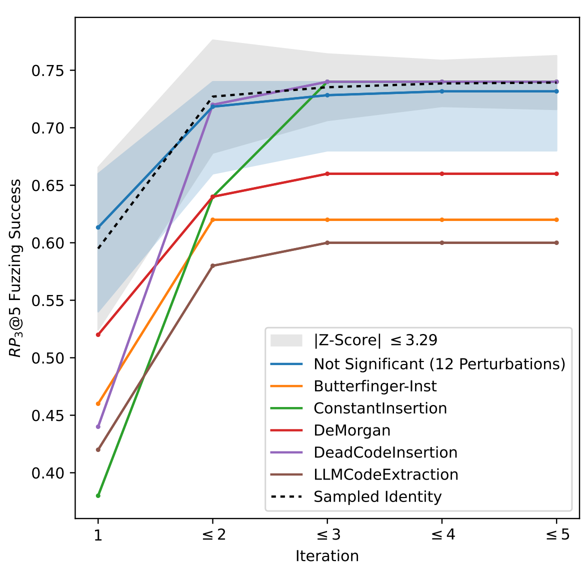


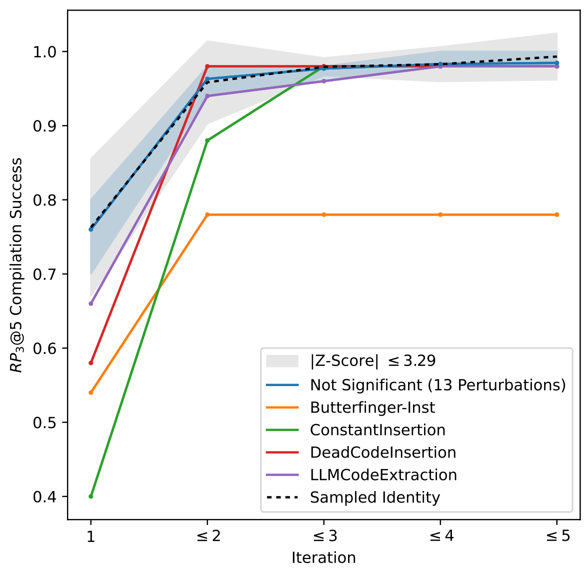
Figure 7.5: Stacked correctness results for the non-robust perturbations of iteration one and iteration five, distinguishing between *compiler*, *clippy*, and *fuzzing success*.

identified. The compiler output looked quite different among the failing files. Nevertheless, the compiler errors resulted from multiple issues with types and basic syntax issues, which are also mostly quite easy to fix with the feedback of the compiler. Therefore, this could also explain why this perturbation did not cause significant outlying performance in later iterations.

Consequently, the *feedback loops* improved the robustness under *stochastic perturbation* strategies. Since the iterations fixed the deficiencies with **ConstantInsertion** and **DeadCodeInsertion**, the general robustness after all generations is improved, because fewer perturbations produce outlying performance. However, both improved perturbations produced failures that are either because of compiling or linting errors, showing that these failures might be easier to fix with *feedback loops*. Figure 7.6 illustrates the improvement progress of the outlying perturbations of iteration one. It gets clear that as soon as the *compilation success* is fixed in iteration ≤ 2 (**DeadCodeInsertion**) or ≤ 3 (**ConstantInsertion**), also the *fuzzing success* is elevated into the expected noise area. Yet, as presented



(a) *RP_3@5* for *fuzzing success* across the *feedback loop* iterations.



(b) *RP_3@5* for *compilation success* across the *feedback loop* iterations.

Figure 7.6: Lineplot showing *RP_3@5* values for the different iterations under *stochastic perturbations*. The perturbations get *Z-Score* classified in *iteration one*, and four outlying strategies are represented with a unique line. In addition, a dashed line shows the sampled **Identity** baseline and the area of nondeterministic noise, according to the absolute *Z-Score* being ≤ 3.29.

in the previous Chapter 5, the model does not fix the other two perturbations.[2]

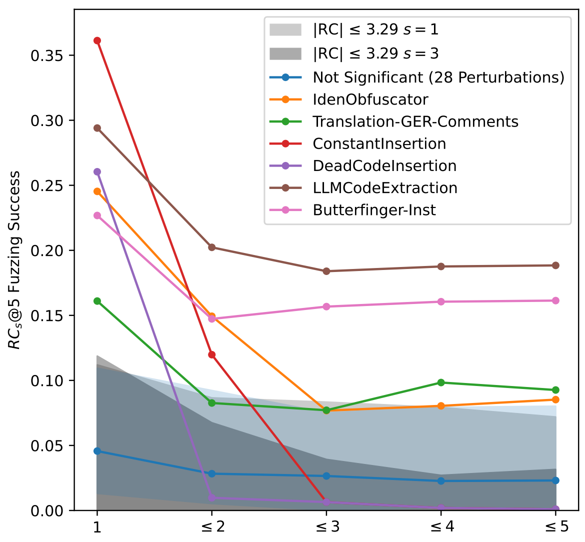
While for **Butterfinger-Inst** this is negligible, because of the drastic modifications in the intent of the prompt, it is interesting that the model does not sufficiently translate **LLMCodeExtraction** perturbations. Figure 7.5 shows, that under **LLMCodeExtraction** the model had a few problems compiling and linting. However, while *compilation success* already was significantly better than for **DeadCodeInsertion**, the model produced worse *fuzzing success*. Recalling Chapter 5 with all iterations, the model improved the *compilation success* of **LLMCodeExtraction** to 0.98 and *fuzzing success* to only 0.6. So while the iterations significantly improved the likelihood for a successful compilation and linting, the model did not manage to fix enough *fuzzing counterexamples* to produce a robust performance for this perturbation. This suggests that *feedback loops* are more likely to improve robustness when the underlying issues are primarily due to compiler or *clippy* failures. This can be summarized for RQ3 considering both *deterministic* and *stochastic* perturbations in the upcoming section.

7.5 Discussion in the Context of RQ3

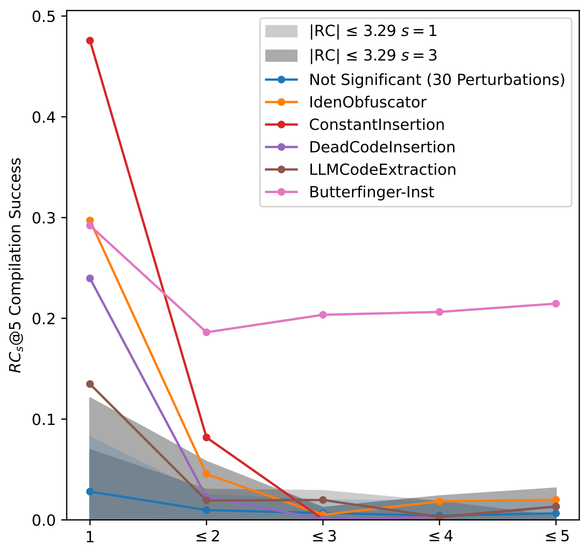
In the following, we discuss these observations in detail, showing why certain errors are systematically easier (or harder) to correct and what this means for RQ3. The discussed results present the necessary information to answer RQ3 for *GPT-4o-mini*.

As the focus is only on the impact on robustness, it is possible to combine the translation

[2]After iteration ≤ 2 it is observable that some previously not significant perturbation breaks out of the *Z-Score*, namely **LLMVariableImprove**.



(a) RC_s@5 for *fuzzing success* across the *feedback loop* iterations.



(b) RC_s@5 for *compilation success* across the *feedback loop* iterations.

Figure 7.7: Lineplot showing RC_s@5 values for the different iterations under all perturbations. The perturbations get *Z-Score* classified in *iteration one*, and five outlying strategies are represented with a unique line. In addition, there are two areas, one for $s = 1$ and one for $s = 3$, according to the absolute *Z-Score* being ≤ 3.29.

results for *deterministic* and *stochastic* perturbations. While this is not possible with $RP_s@k$, because different s present different difficulties and are thus hard to compare, it is feasible with *robust change*, as this metric presents the relative performance change compared to the **Identity** baseline. As long as for $s = 1$ the baseline RP_1@5 is taken, and for $s = 3$, RP_3@5, the relative amount of change can be compared.

Figure 7.7 shows the progress of this metric across the iterations. This figure shows that the *feedback loops* influence the robustness of the model. It highlights that with iterations, the change gets noticeably less drastic. Especially for *compilation success*, the model improves the amount of change significantly, except for the negligible **Butterfinger-Inst**. However, on *fuzzing success*, this effect is only strong enough for two perturbations **ConstantInsertion** and **DeadCodeInsertion**. While the other perturbations also yield less change than in iteration one, they are still definitive outliers. Thus, while *feedback loops* notably reduce the severity and frequency of robustness deficits, especially regarding *compilation* and *linting* errors, they rarely eliminate deeper logical, fuzzing-related robustness issues completely. Specifically, the non-robust perturbations that were fixed with the iterations resulted primarily from issues in *compilation* and linting.

IdenObfuscator as well as **LLMCodeExtraction** improved in *compilation success*, without leading to a strong enough effect in *fuzzing success*, therefore highlighting that *feedback loops* did not manage to fix most of the non-fuzzable translations.

This can also be seen in Figure 7.8. In the first iteration, the major reason for failure is *compiler* errors. This changes significantly in the next iteration, where most *compiler failures* can be fixed. Therefore, the total amount of failure decreases, but the amount of *fuzzing failures* increases. As some of the previously failing compilations are now accessible in regard to fuzzing success. However, it shows that even in iteration five, the number of

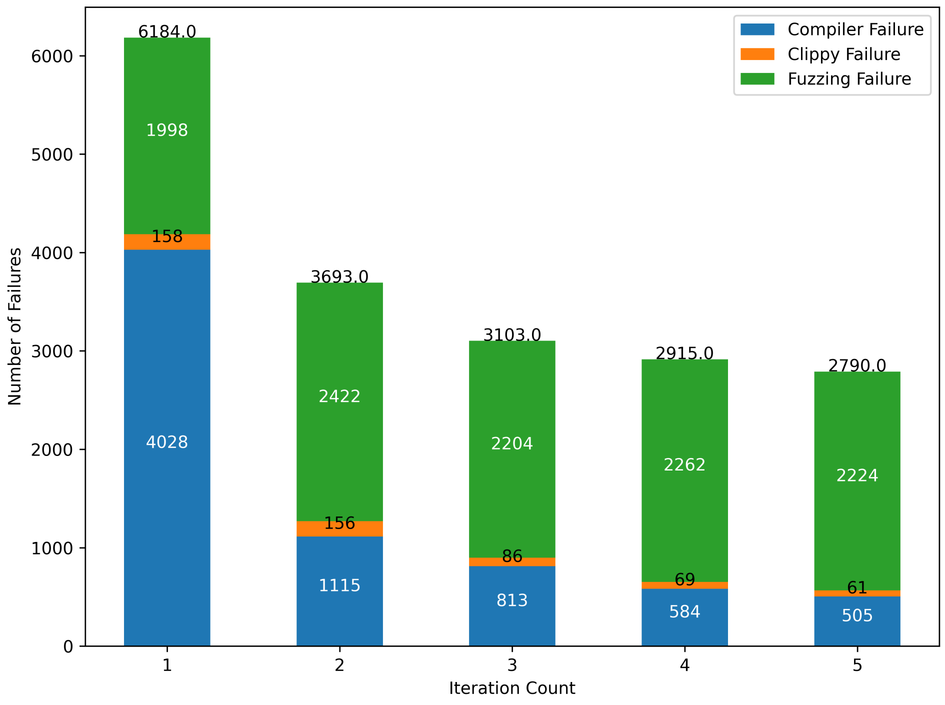


Figure 7.8: Detailed analysis of which check fails the translation task for each iteration of the *feedback loops*.

fuzzing failures could not be drastically reduced. This supports the finding that *feedback loops* might work best on *compilation* and *clippy failures*.

This is a reasonable finding, as *compiler* or *clippy* outputs directly explain the issue and its position, which makes it easier for the model to understand the problem and produce a repaired translation. In case of *fuzzing failure*, the model only receives the specific counterexample that failed. Consequently, the model has to refine the logic behind the translation which is indeed harder than fixing a compiler or linting error [Cod24].

What does this mean for robustness? The *feedback loops* have a good chance of improving robustness, especially if robustness deficiencies are a result of *compilation failures* or *clippy failures*. The strategy might also improve some *fuzzing* related deficiencies, yet the chance of improvement is much lower. Furthermore, as the translation system also improves the baseline performance with iterations, a perturbation that causes fuzzing deficiencies cannot improve the same as the baseline does, making it harder to produce a robust translation for this perturbation. Lastly, non-robust perturbations that cause improved performance can only become robust in higher iterations if the amount of improvement per iteration is less than the baseline's improvements. From a practical perspective, *feedback loops* are highly beneficial as they consistently enhance model performance and notably reduce robustness deficits, making them valuable for real-world code translation systems.

In conclusion, *feedback loops* effectively improve the robustness of *GPT-4o-mini* against certain perturbations, particularly those causing explicit *compilation* or *linting* issues. However, their ability to handle more complex fuzzing failures remains limited, which could be a consideration for future improvements.

8 Exploring the Correlation between Semantic Similarity and Robustness

Prior chapters examined the robustness of *GPT-4o-mini* under various perturbations, with and without *feedback loop* iterations. These investigations highlighted certain perturbations that tend to produce non-robust behavior and offered preliminary insights into why specific perturbations might lead to larger performance changes. However, the *magnitude of change* itself has not yet been incorporated in these interpretations.

8.1 Motivation in Measuring Similarity

Intuitively, one might expect that perturbations that substantially modify the **Identity** would be more likely to trigger non-robust behavior. Whether this intuition holds true, however, has remained an open question in this thesis and also in related work. To capture such variations, this chapter employs *cosine similarity* on embedding vectors as an indicator for semantic similarity. These embedding vectors are produced by *OpenAI's ada-002* embedding model [Ope22c].

This leads directly to RQ4, investigating whether lower *semantic similarity* correlates with greater robustness deficits. By leveraging the findings from previous experiments on *deterministic* and *stochastic* perturbations, the subsequent sections analyze to what extent embedding-based similarity can predict or explain non-robust outcomes.

With the *cosine similarity* on embedding vectors, the framework can explore whether files that diverge more from the original trigger stronger performance deviations. The *cosine similarity* approach is practical because modern language models typically produce embeddings that capture various semantic features of the text. Rather than focusing on purely lexical distances, this approach checks if the model itself encodes two code snippets as similar.

If the results signal a clear correlation between *cosine similarity* and translation success, this would present a strategy for predicting robustness for certain perturbations. Furthermore, this would make it possible to filter out perturbations that lead to overly drastic changes, as they might bias the robustness evaluation.

8.2 Similarity Baseline

To get a better understanding of the values produced by *OpenAPI's ada-002*, a baseline distribution was created. As explained in Section 4.4.4, this distribution is derived from pairwise comparisons of all 50 *C* files in the benchmark dataset. Prior UMAP visualizations and Figure 5.1 indicated that the files mostly differ noticeably from one another. As a result, the computed similarities capture how the embedding model interprets semantically different files that share only the general property of being *C* code.

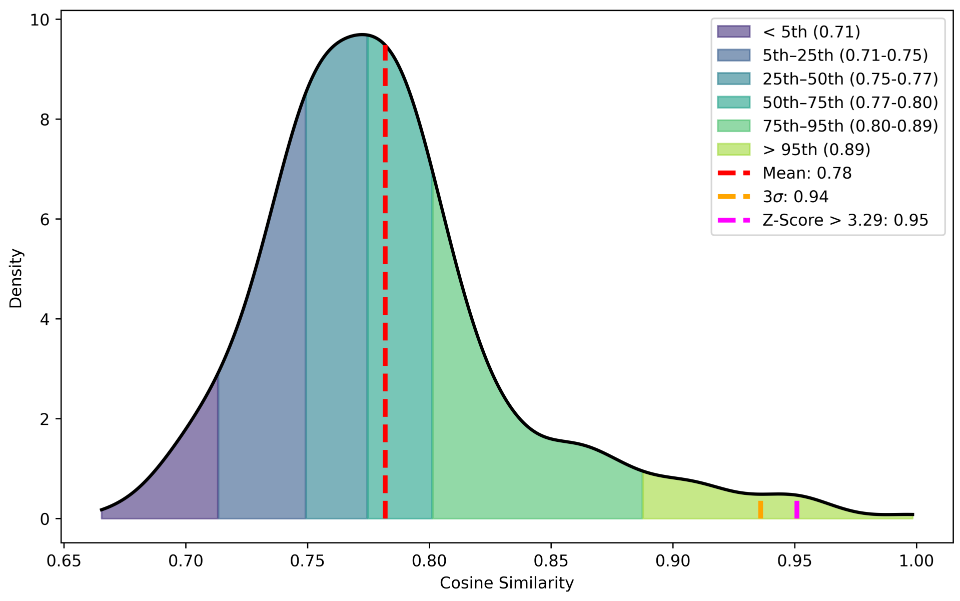


Figure 8.1: KDE density plot [Węg18] of the *cosine similarity* distribution among pairwise comparisons of the benchmark dataset with colored percentile intervals. The red dashed line represents the mean, the orange the three sigma rule (Z>3), and the magenta dashed line marks the *Z-Score* outlier threshold ($Z > 3.29$).

Figure 8.1 visualizes the calculated distribution of *cosine similarity* values. It shows that not a single comparison falls below 0.65. It appears that the model consistently classifies all files as at least somewhat similar, presumably because they share common programming elements. Considering that most files are semantically different, the value range of minus one and one is not accurate for predicting that a model might show non-robust behavior, and justifies why such a baseline assessment is necessary.

As detailed in Section 4.4.4, to specify similar perturbations, the *three sigma rule* can be used. Employing this rule one the presented examples in Section 4.4.4 would classify the bitwise sum in Listing 4.3 and the instruction that modified *Rust* to *Tyst* as semantically too different. This demonstrates that the proposed strategy is able to classify the examples as desired and sparks the question of whether we can observe a similar behavior for the significantly non-robust perturbations.

8.3 Deterministic Perturbations

To address RQ4, we compute the *cosine similarity* between each *deterministic perturbation* and the **Identity**. Figure 8.2 displays these similarities both with and without *feedback loops*. Perturbations left of the black line are considered less meaningful, whereas those to the right meet the similarity threshold. This goes back to the initial motivation behind this component. Perturbations that are highly different from the **Identity** are positioned on the left side, and subsequently considered less meaningful. Each side is further divided into robust and non-robust regions using the *Z-Score* threshold of 3.29 on the *Sampled Identity* $\boldsymbol{RP}_1$@5 distribution. The yellow areas present outliers that produced higher success than the expected noise, while red regions indicate significantly lower *fuzzing success*. In this way, the plot provides a comprehensive robustness evaluation by integrating all previously discussed information with similarity scores.

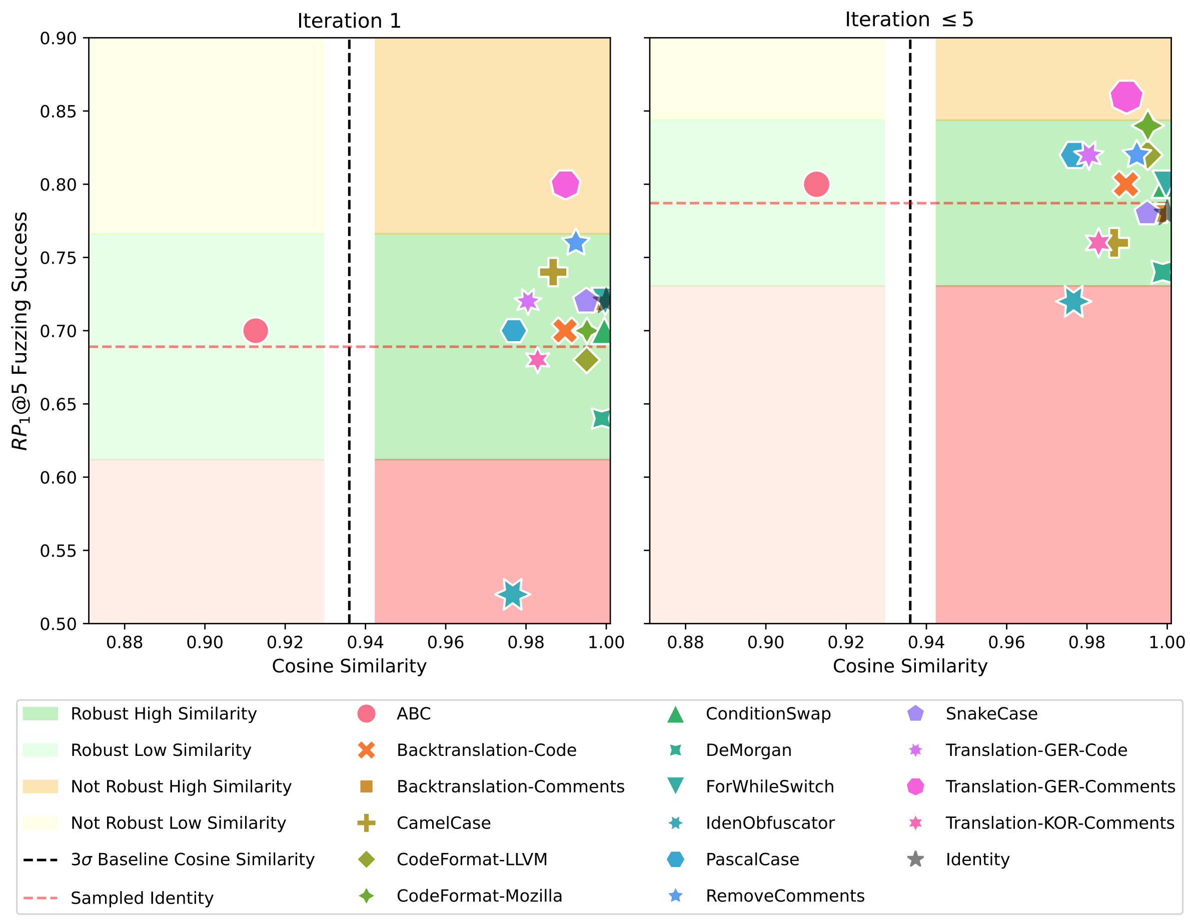


Figure 8.2: *Cosine Similarity* and *fuzzing success* of *deterministic code perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

A model with multiple perturbations in the dark red area (bottom right) is deemed non-robust, as it produces significantly worse results on similar inputs. Likewise, numerous perturbations in the dark yellow area signal inconsistent performance, although unexpectedly strong outcomes here are less concerning in practice. An ideal robust model would exhibit perturbations exclusively in the green area. In addition, robust performance on the right is anticipated, while robust performance on the left is even more noteworthy, because of the lower similarity to the **Identity**. Finally, if a model fails to achieve robust results for low-similarity (left-side) perturbations, this should not overly penalize its overall robustness evaluation.

Focusing on the visualized information in Figure 8.2 confirms the previous findings that *GPT-4o-mini* primarily produces robust results and that *feedback loops* can reduce the impact of non-robust deficits (**IdenObfuscator** and **Translation-GER-Comments** move closer to the baseline with *feedback loops*). Furthermore, the plot details that the majority of perturbations produced similar files and subsequently adhered to the semantic similarity requirement. However, the **ABC** perturbation is on the left side of the three sigma baseline, noting less similarity, yet the model managed to produce a robust result. The UMAP in Figure 5.2 already suggested that **ABC** is the deterministic perturbation with the most amount of change, yet the expectation was that it was drawn to other files with equivalent identifier names after perturbation. The *cosine similarity* plot shows that this was not the only reason, because for the embedding model, the changes of this perturbation were the strongest for all *deterministic perturbations*.

Depending on the view, this could show boundaries of the not-so-optimized *cosine similarity* approach. There is no ground-truth on what should be considered similar and what should not. Yet the expectation would have been that the model interprets two files only differing in identifier names as similar. Clearly, one could use another strategy to draw the *cosine similarity* baseline threshold, e.g, by orienting on the 95th percentile at 0.89 (see Figure 8.1). However, the **ABC** perturbation would still significantly show the lowest similarity to the **Identity**. Nonetheless, the *cosine similarity* can be considered as the model's own "opinion" on the similarity of perturbations. Since embeddings are fundamental to how LLMs represent and process information, the *cosine similarity* might better capture the model's perceived change compared to human annotations. This leads to the interpretation of the correlation between *cosine similarity* and robustness results.

With **ABC** being the least similar and yet a robust perturbation, this already presents that the similarity alone is not the driving factor for worse robustness. Furthermore, the most difficult perturbation is **IdenObfuscator**, yet the *cosine similarity* does not suggest that perturbations produced overly different files. The interpretation in Section 6.1 concluded that the difficulty of this perturbation may be caused by the unusual identifier names that could mislead the embeddings. Comparing **ABC** and **IdenObfuscator**, this hypothesis can not be confirmed, as **ABC** misled the embeddings more and still resulted in robust translation success.

Focusing on perturbations with a *cosine similarity* very close to 1.0 might highlight that the model's performance is more clustered around the *Sampled Identity*, and with decreasing similarity, the performance begins to spread. This is especially visible without the *feedback loops*. Yet there are also exceptions like the **DeMorgan** perturbation, which is suggested to be very similar, but almost negatively crosses the robustness threshold. To give a concise assessment of the correlation, the *Pearson correlation coefficient* [SBS18] is used. As robustness shows by both performance decrease and performance increase, the $\boldsymbol{RC}_1@5$ metric is specifically chosen for the calculation. The *Pearson correlation coefficient* is a well-established correlation metric whose value ranges are commonly mapped to textual descriptions. While Schober et al. [SBS18] point out that judging correlation only with fixed-range correlation intervals is not optimal, the thesis uses their table that classifies correlation values into textual descriptions. The combination of the visual analysis and the coefficient gives a clearer picture of the correlation and mitigates the concern of misinterpreting the coefficient.

Without the *feedback loops*, the coefficient between *cosine similarity* and $\boldsymbol{RC}_1 5@$ is -0.0103, and with *feedback loops* it is 0.0193. According to Schober et al. [SBS18], this presents weak to no correlation. So heuristically, there is no noteworthy correlation between similarity and robustness for the *deterministic* perturbations on code.

This may again suggest that robustness to perturbations is highly individual, but could also reflect that the used similarity measure is not usable for quantifying the similarity between code files.

Figure 8.3 shows the same plot for perturbations on the instruction. Recall that for instructions, there is no particular baseline, and the values of the code comparison are used. The figure details that the translation into German or Korean presents a significant change. This suggests that the *cosine similarity* baseline on code is not the optimal approach for deciding whether a perturbation on an instruction is meaningful or not. This might be

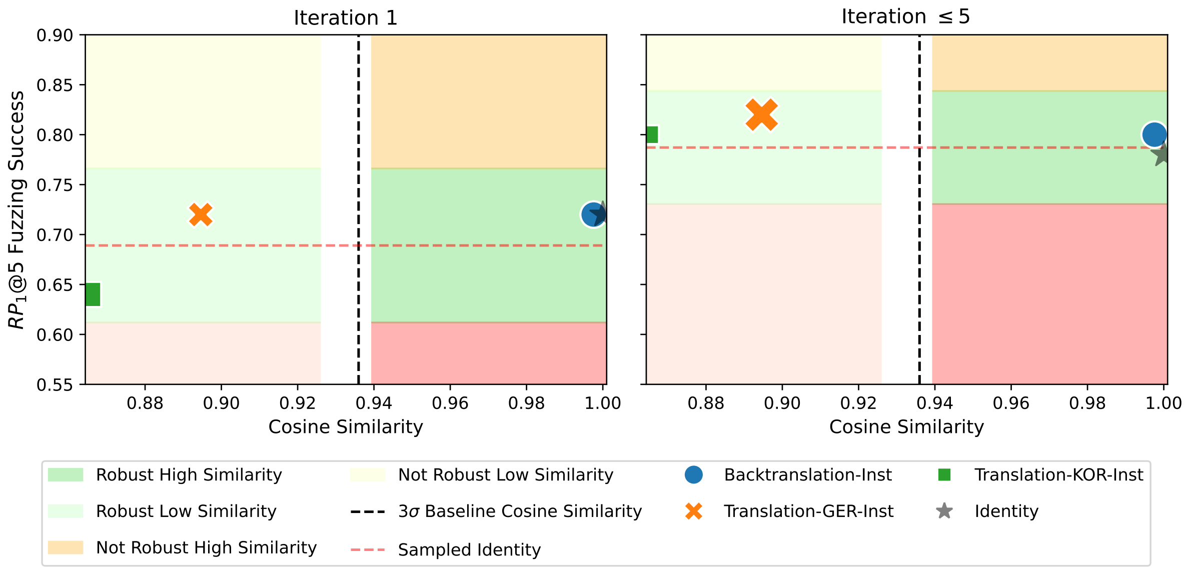


Figure 8.3: *Cosine Similarity* and *fuzzing success* of *deterministic instruction perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

caused by the fact that the average token count per code file is significantly larger than for the instruction. The **Identity** instruction: "Translate the following C code to Rust. Keep all identifiers exactly as they are." has a token count of 16. It seems reasonable that changing or adding some tokens might stronger effect on the *cosine similarity* of this sentence.

While the classification of meaningful perturbations can be refined for instructions, the *cosine similarity* results can be used for the correlation coefficient. Without *feedback loops*, the model shows a correlation coefficient of -0.672 (moderate) and with -0.306 (weak). So, for perturbations on instruction, such correlation could also not clearly be found. However, the correlation might suggest that the *feedback loops* can improve robustness issues for less similar instruction perturbations. However, the experiments only involved three data points. This makes such correlation analysis less trustworthy.

In addition, it is important to note that correlating translation success with similarity has to be taken with caution, especially with $s = 1$, as minor fluctuations are not necessarily a sign of robustness characteristic, but could just be nondeterministic noise. However, when considering the confidently classified non-robust perturbations **IdenObfuscator** and *Translation-GER-Comments*, there is also no definitive correlation between similarity. Furthermore, it might also be noteworthy that there is also no clear correlation between *compilation success* and *cosine similarity*. The plots and correlation values are presented in Appendix A.5.1.

The experiment on *stochastic perturbations* will show if similar findings can be observed for *stochastic* perturbations and the less fluctuating *robust pass* with $s = 3$.

8.4 Stochastic Perturbations

Figure 8.4 details the correlation between *cosine similarity* and *fuzzing success* for *stochastic perturbations* with RP_3@5. Since each *stochastic* perturbation involves three different

Tokens	**DeadCodeInsertion**	**ConstantInsertion**	**Identity**
Mean per File	578.47	542.68	454.54
Total per Dataset	115693	108535	90908

Table 8.1: Comparison of token counts for less similar perturbations.

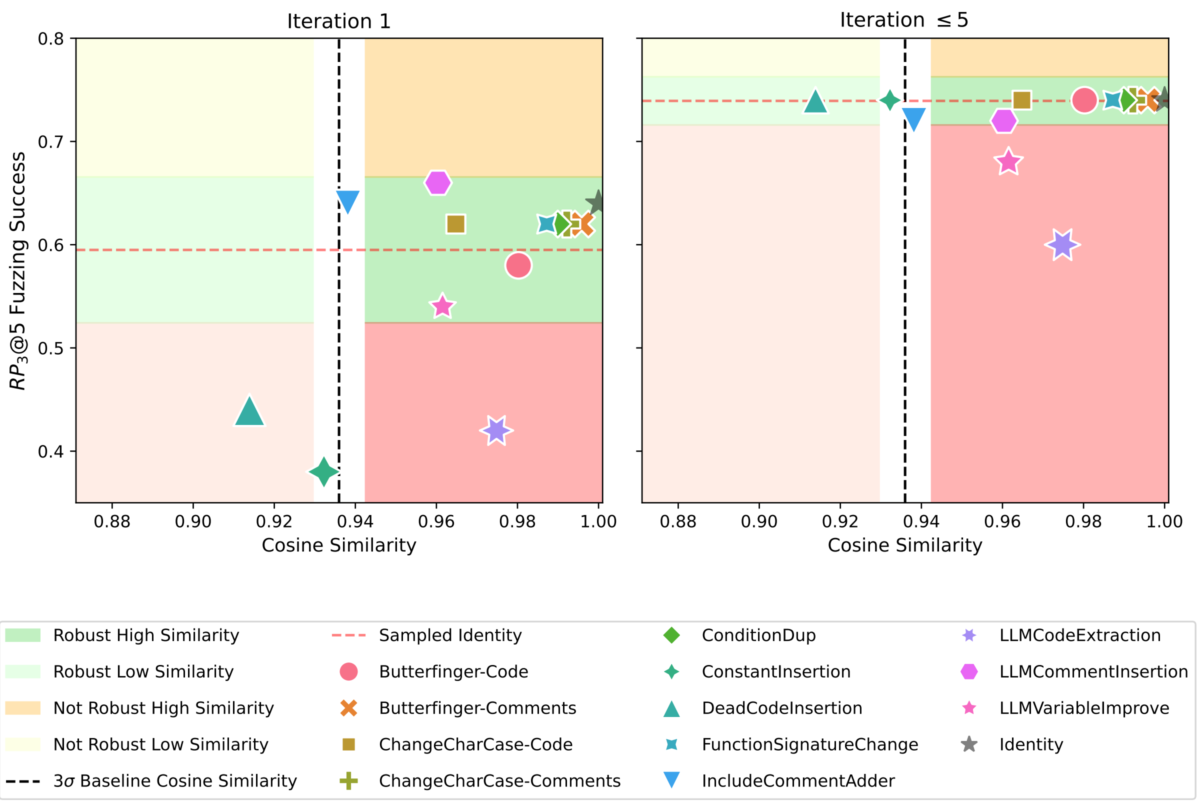


Figure 8.4: *Cosine Similarity* and *fuzzing success* of *stochastic code perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

variations ($s = 3$), there can be three different *cosine similarity* values. However, since $RP_3@k$ uses the worst-case approach, the decision was to show the variation with the worst *cosine similarity*. As a translation must also be correct for the least similar variant, showing the least similar and presumably hardest variation allows a better interpretation in regard to correlation.

Comparing the plot without *feedback loops* on the left with the one with *feedback loops* highlights the previously discussed findings, that the *feedback loops* can indeed improve robustness, especially for the perturbations that fail because of *compilation success* (**DeadCodeInsertion** and **ConstantInsertion**).

In addition, the embeddings suggest that **DeadCodeInsertion** is the least similar *stochastic* perturbation on code, followed by **ConstantInsertion**. Since **DeadCodeInsertion** and **ConstantInsertion** cross the three sigma threshold, they can be deemed as less meaningful for the evaluation. However, they are rather close to the threshold and therefore not classified with high confidence. Nevertheless, it seems reasonable that the embedding for **DeadCodeInsertion** and **ConstantInsertion** present larger change, as

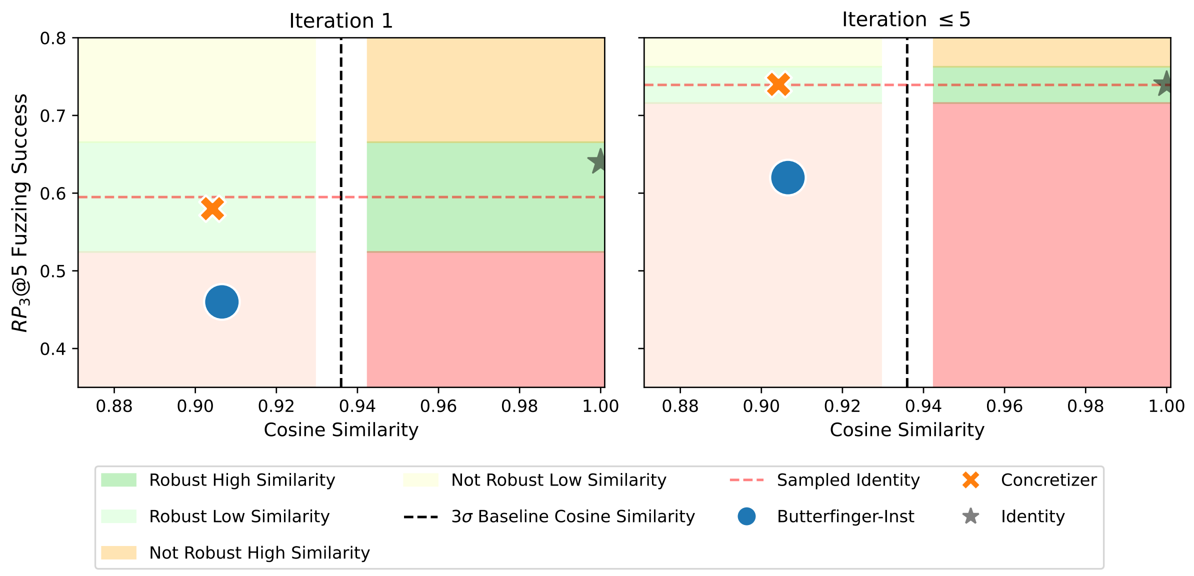


Figure 8.5: *Cosine Similarity* and *fuzzing success* of *stochastic instruction perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

they drastically increase the file length, by adding code snippets (see Table 8.1). The UMAP in Figure 5.3 already showed that **DeadCodeInsertion** might have produced a significant change. Yet, the interpretation was similar to **ABC**, and it was thought that this is caused by multiple perturbed files having similar code snippets and subsequently higher similarity. Figure 8.4 and Table 8.1 details, that this was not the only reason.

In iteration one, the figure shows that perturbations with very high similarities show less spread in the *fuzzing success*, as those that have less similarity. With *feedback loops*, this pattern vanishes, as the originally failing **DeadCodeInsertion** and **ConstantInsertion** translations get fixed and other perturbations move closer to the baseline. The *Pearson correlation coefficient* [SBS18] on RC_3@5 and *cosine similarity* confirms this finding. Without *feedback loops* there is a correlation of -0.714 (strong), and with 0.023 (weak). Consequently, this is the same behavior one could find for *deterministic perturbations* on instructions. However, under *stochastic perturbations*, this correlation is even higher. In addition, *compilation success* shows similar correlation values (see Appendix A.5.1).

Besides that, it is interesting that the most difficult perturbation with *feedback loops* **LLMCodeExtraction** is considered quite similar. As shown earlier, the perturbation increases complexity by adding nested function calls. However, this does not strongly affect the *cosine similarity*. This might as well show that *cosine similarity* is not the best approach to measure perturbation similarity on code.

Figure 8.5 visualizes the *cosine similarity* and *fuzzing success* for *stochastic perturbations* on instructions. This highlights again that the three-sigma baseline on the code files might not be the best way for classifying the relevance of instruction perturbations. Furthermore, it confirms once more that the *cosine similarity* is more sensitive when the **Identity** involves only a small number of tokens. As there are only two data points, the correlation coefficient is not that meaningful. It suggests a correlation of 1.0 for both iterations one and ≤ 5. This is clearly because the less similar **Concretizer** has less RC_3@5 than the more similar **Butterfinger-Inst**. While the correlation is not meaningful, it is still surprising that the **Concretizer** presented more change to the embedding. The suggestion would have been that **Butterfinger-Inst** shows clearly that it is highly dissimilar, as it tends to modify the valuable target language part of the instruction. Since **Concretizer** only appends text and **Butterfinger** only modifies existing text, this leaves the impression that the length difference also has a great impact on *cosine similarity*, even when the added text adheres to the original semantics.

Overall, the results on *stochastic perturbation* suggest that the correlation between *cosine similarity* and translation success with *feedback loops* is not that prominent, as one would have expected. However, the perturbations on code leave room for the interpretation that *feedback loops* positively influence non-robustness for less similar inputs. There was a strong correlation between less similarity and higher $\boldsymbol{RC}_3$@5. However, this strong correlation is primarily driven by the two failing perturbations **DeadCodeInsertion** and **ConstantInsertion**, which other interpretations showed might be easily fixable, because they are a result of compilation or linting errors. Excluding these perturbations, the correlation is weak without (-0.177) and nonexistent with *feedback loops* (-0.005).

8.5 Interpretation for RQ4

This section returns to RQ4: “What is the correlation between semantic similarity and perturbation-based robustness?”

The initial expectation was that more similar inputs, showing in higher *cosine similarity*, would yield fewer robustness issues. This led to the idea that *cosine similarity* might be leveraged as an a priori predictor for the difficulty and relevance of certain perturbations.

The sections of this chapter analyzed this correlation between *cosine similarity* and translation success for *deterministic* and *stochastic* perturbations. Both *deterministic* and *stochastic* perturbations revealed that the correlation between *cosine similarity* and *fuzzing success* (or *compilation success*) was either very weak or nonexistent when *feedback loops* were applied. Certain subsets like *stochastic perturbations* on code, or *deterministic perturbations* on instructions, suggested that *feedback loops* might impact this correlation. Without *feedback loops*, these subsets exhibited moderate to strong correlation, indicating that higher similarity tends to reduce the deviation in translation success compared to the *Sampled Identity* baseline. This effect vanished when *feedback loops* were employed. This might suggest that *feedback loops* can act as a fallback to mitigate initial errors, which makes any direct correlation between similarity and robustness less apparent.

However, it was noted that the strong correlation was mostly due to the two easily fixable perturbations **DeadCodeInsertion** and **ConstantInsertion**. Thus, for a better assessment, we can combine the data points of both *deterministic* and *stochastic* perturbations on all targets. Calculating the *Pearson correlation coefficient* for this set only yields a moderate correlation of -0.486 without *feedback loops* and a weak correlation with *feedback loops* -0.064. So *feedback loops* still reduce the correlation, yet the overall value with all data points is less significant. Considering that the model involves nondeterministic fluctuations, the correlation becomes even more insignificant.

Consequently, contrary to the initial hypothesis, a clear linear relationship between embedding-based similarity and translation robustness could not be observed. One could think that this could be caused by comparing aggregated *cosine similarity* and aggregated $\boldsymbol{RC}_s$@5 values for the files of the dataset, which could flatten certain correlations. However, computing the correlation coefficient between the files *cosine similarities* with their $\boldsymbol{RC}_1$@5 values yielded even less correlation.

Therefore, there are several potential reasons why no clear correlation could be found. The first reason could lie in the impact of token-level similarity versus global similarity. Small changes to critical tokens (e.g., modifying the target language "Rust") can result in disproportionately large performance drops, whereas identifier renaming might have negligible effects (**ABC**). This indicates that the translation process may be more dependent on certain keywords and is highly individual to the actual task than the overall textual similarity measured by embeddings. Consequently, large textual modifications might not affect translation success, while smaller, but pivotal token perturbations could drastically impact translation outputs. The significance of pivotal tokens for LLM performance is also highlighted in [Abd+24].

Another reason for not finding clear correlations could be limitations of the embedding model. According to *OpenAI* [Ope22c], the chosen model *ada-002* was optimized to unify multiple tasks like text search, text similarity, and code search into a single embedding model. Although it incorporates semantic code-search tasks, it may not fully capture the syntactic, structural, or even logical aspects of source code. Lastly, the nondeterministic nature of LLMs leads to the model producing fluctuating outputs for exactly similar inputs, which makes it harder to show a definitive correlation.

Combining these factors, the conclusion is that higher *cosine similarity* alone does not reliably predict higher robustness. While embedding-based methods provide interesting insights into overall text similarity, further research using code-specific approaches may be better to more accurately determine which perturbations truly represent challenging shifts in semantics. However, it may also just be the fact that the translation robustness of a model is not predictable, as there are only minor factors or randomness that determine translation success. Hence, RQ4 must be answered negatively: With the proposed methodology, there does not seem to be a clear correlation between *cosine similarity* and robustness.

9 Extending the Analysis to Multiple Models

The previous chapters provided a comprehensive robustness evaluation for *GPT-4o-mini*. With the proposed framework, the interpretations showed that this model itself is quite robust, with only a few Perturbations that caused significant performance changes. Moreover, it could be shown that the *feedback loops* can improve, but do not ensure robustness. Lastly, no clear correlation between *cosine similarity* of perturbations and *robustness* could be observed for all scenarios. This chapter aims to clarify whether all these findings are consistent among other LLMs. Specifically, the chapter aims to bring evidence for discussing RQ5: "Are robustness results consistent using different LLMs?"

To answer this question, the thesis takes the same approach as it did for *GPT-4o-mini*. That means it begins by defining a baseline and analyzing the performance differences under perturbations. While for *GPT-4o-mini* the experiments involved *deterministic* and *stochastic* perturbations, the upcoming models (i.e., *GPT-3.5-turbo, Phi-4, Qwen2.5-Coder*) are only analyzed under *deterministic* perturbations. This is because of higher costs in price and runtime, explained in Section 5.1. Furthermore, to reduce the complexity of the chapter, the focus is mostly only on the more important *fuzzing success*. After discussing the robustness with *feedback loops*, the chapter compares results without *feedback loops* to validate whether the approach always leads to increased robustness. Lastly, the correlation between *cosine similarity* and robustness for all LLMs is investigated.

9.1 Motivation

With the previous analysis, it could be shown that the proposed framework enables a comprehensive robustness evaluation. This led to certain conclusions for the evaluated *GPT-4o-mini*. By performing an evaluation on other models, it can be examined whether the framework can be easily utilized to evaluate the robustness of other LLMs. This provides more information to the main RQ: "What methodologies and components should be integrated into a comprehensive evaluation framework to assess the robustness of an LLM-based code translation system?"

Furthermore, by performing an evaluation on other models, the thesis examines whether prior conclusions about robustness can be transferred to other LLMs or whether they are individual to *GPT-4o-mini*. This examination presents the evidence for answering RQ5.

To measure the robustness, it is necessary to understand the baseline performance of the different LLMs.

9.2 Baseline and Noise across Different LLMs

The baseline assessment follows multiple motivations in this chapter. At first, it should compare the general translation performance for the different LLMs, to show which model

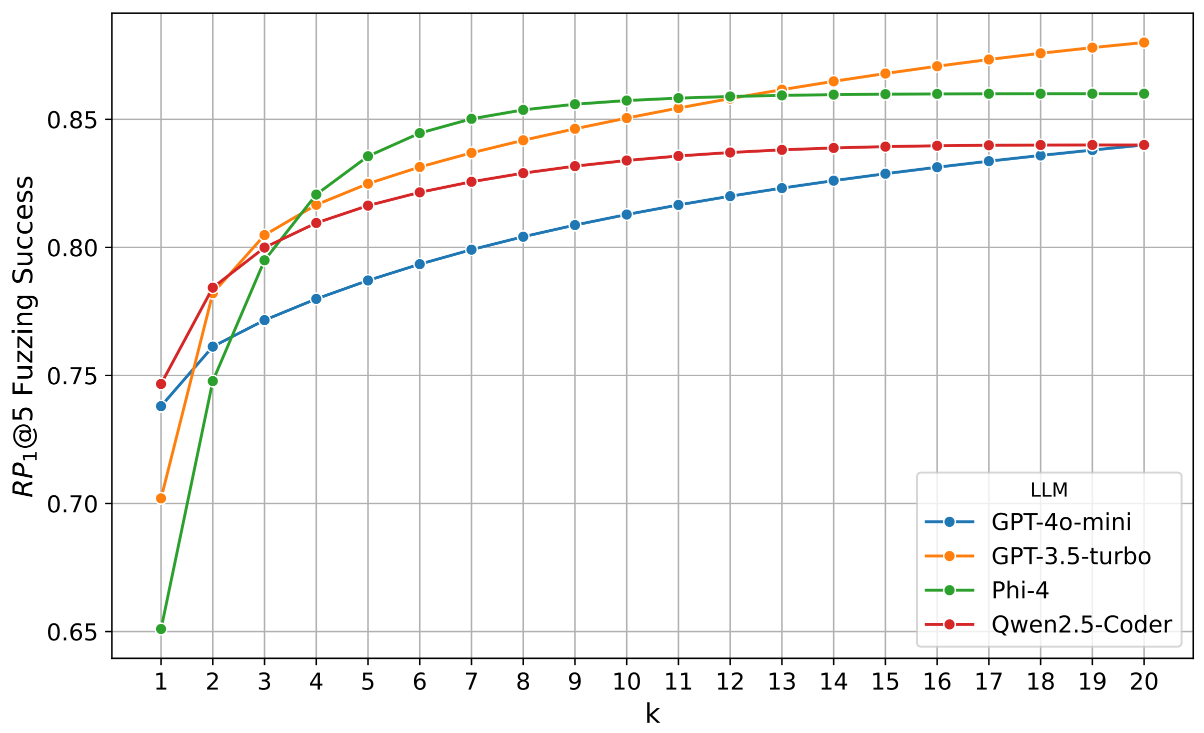


Figure 9.1: $RP_1@k$ for different k values of all 20 **Identity** runs.

produced the best results for the task of translating *C* to *Rust*.

The second motivation is to get insights into the expected baseline for the robustness assessment. That means that the previous strategy is applied, where all different groups of five are built, to get a mean $RP_1@5$ for $n = 5$. This enables the assessment of nondeterministic noise to later decide what changes are caused by non-robustness and what could be nondeterministic noise, which is the third motivation of this section.

To keep coherence and reduce complexity, the baseline analysis is presented on the code translation system with *feedback loops*. This makes the results more comparable to those that were presented in Section 5.2.3.

Recall that the baseline performance for *GPT-4o-mini* was assessed by incorporating $n = 20$ runs. These runs resulted from the experiments performing five runs per s on *deterministic* ($s = 1$) and *stochastic* ($s = 3$) perturbations, which in combination yielded twenty runs for the **Identity**.

However, since the other models are only evaluated under *deterministic* strategies there were fifteen extra runs only on **Identity**, to create a comparable baseline with 20 runs[1].

9.2.1 Translation Performance Comparison

Figure 9.1 illustrates the *fuzzing success* for all models over the $n = 20$ runs with different k. This plot is different from the previous baseline assessment, which focused on the *Sampled Identity* $RP_1@5$ performance for all possible groups of five that can be built out of these twenty runs. This plot directly incorporates all runs into the calculation of the $RP_1@k$ metric.

The figure highlights multiple interesting points. First, it suggests that the selection

[1]For *Qwen2.5-Coder* it took nineteen extra runs, as there is only one run under deterministic perturbations.

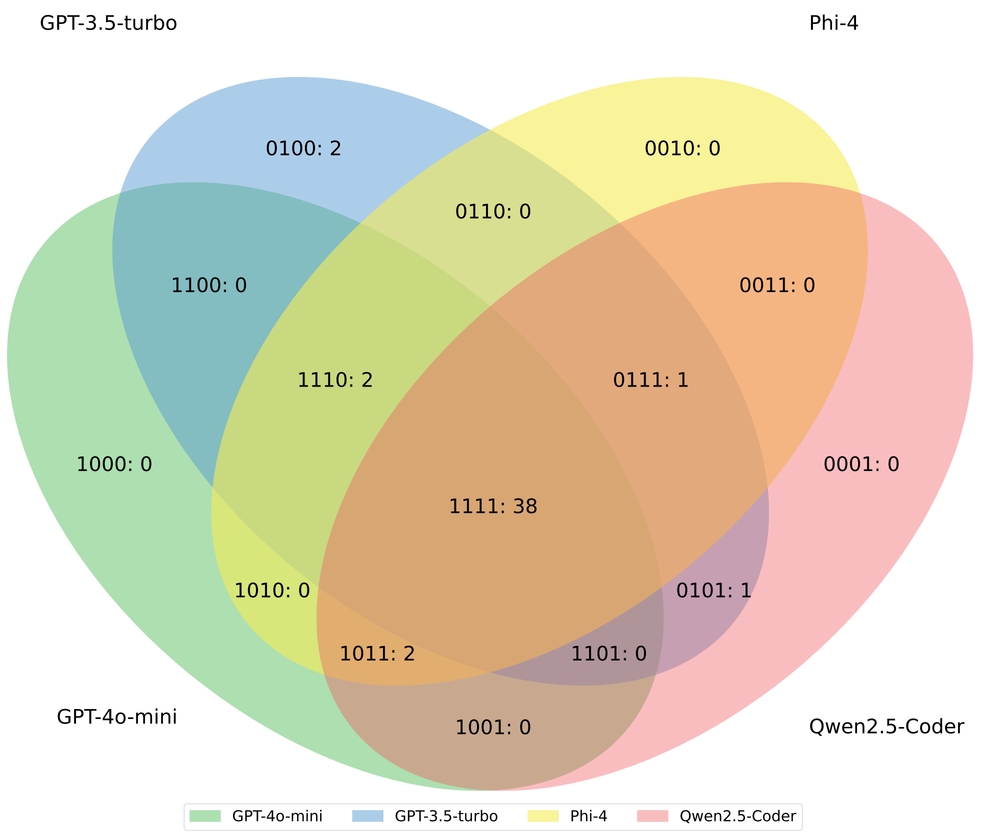


Figure 9.2: Venn diagram [Ven80] of the solved files over twenty runs for the different models. The diagram includes a binary encoding, specifying what values belong to what combination of models, to simplify interpretability. The exact order of the *binary encoding* is *GPT-4o-mini*, *GPT-3.5-turbo*, *Phi-4*, *Qwen2.5-Coder*.

of the best model depends on how many runs one is willing to perform. For one-shot results, the coding model *Qwen2.5-Coder* seems to be the best choice, directly followed by *GPT-4o-mini*. Increasing the number of runs shows a steep incline for the performance of *Phi-4*, which performs best for the range of four to eleven runs, where it then begins to saturate. Such saturation can also be observed for *Qwen2.5-Coder*, yet, however, with a stagnation at worse performance than *Phi-4* and *GPT-3.5-turbo*. Neither *GPT-4o-mini* nor *GPT-3.5-turbo* shows a saturation in twenty runs, which leads to *GPT-3.5-turbo* showing best performance amongst all models, when there are more than thirteen runs. As *GPT-4o-mini* shows a small slope in early runs, it generally performs worse than other models, but the figure suggests that with more than twenty runs, this could change.

The *GPT* models not saturating suggests that these models can produce new solutions also after various runs, which could be interpreted as a sign of creativity. Moreover, the figure suggests that the expected translation performance correlates with the model's parameter size. *GPT-4o-mini* as the smallest model, overall produced the worst results, and *GPT-3.5-turbo*, most likely being the largest model, produced the best results.

The *Venn diagram* [Ven80] in Figure 9.2 shows the number of solved files over all twenty runs of the four different models. It highlights that *GPT-3.5-turbo* was able to successfully translate two files, which no other model managed to translate successfully (*0100* in the Figure). However, the plot also points out that with *GPT-3.5-turbo* being the best model, it still misses out on two files that could be translated by all other models (*1011*

$RP_1@5$	GPT-4o-mini	GPT-3.5-turbo	Phi-4	Qwen2.5-Coder
Compilation Success	1.0 ± 0.001	0.999 ± 0.004	0.979 ± 0.005	0.978 ± 0.006
Fuzzing Success	0.787 ± 0.017	0.825 ± 0.034	0.836 ± 0.034	0.818 ± 0.019

Table 9.1: Baseline mean ± standard deviation translation success for all evaluated models among all files utilizing $\text{RP}_s@\text{k}$ for *compilation success* and *fuzzing success.*

in the Figure).

This small digression shows that choosing the right model can influence the overall success, and that the choice of the right model depends on preliminary requirements like price, hardware capabilities, and the acceptable number of experiment runs. The performance under *deterministic perturbations* will show if robustness will also be a factor, which impacts the model choice.

9.2.2 Baseline Performance

To measure the robustness, we need the relevant baseline performance that is used to compare the performance under *deterministic perturbations*. Specifically, this describes the mean $RP_1@5$ among the $\binom{20}{5}$ groups that could be drawn out of the twenty runs. Recall that for *Qwen2.5-Coder*, there is only one run under *deterministic perturbations*, which is reasoned in Section 5.1. That means the relevant baseline for *Qwen2.5-Coder* has to be assessed separately and requires calculating the mean $RP_1@1$ of all $\binom{20}{1}$ groups. However, to have comparable baseline results, this section presents the baseline with $\binom{20}{5}$ and details *Qwen2.5-Coder's* unique one-shot baseline for the robustness assessment at the end of the next section.

Table 9.1 details the values for the different models. Note that the values in Figure 9.1 at $k = 5$ are very similar to the values in the table. One could argue to always work with $k = 5$ and $n = 20$ for the baseline. However, the sampling method comes with the advantage of producing value distribution, which ultimately allows to define the *Z-Score* robustness threshold.

Figure 9.3 visualizes the *fuzzing success* of the different models for the specific files. This figure clearly details why *GPT-3.5-turbo* performed best. The model was able to successfully translate the *incorrect* files of *GPT-4o-mini* (i.e., 18, 23, 43, and 46). Furthermore, it highlights that the other models tend to produce higher values for the *variational* files of *GPT-4o-mini*, which also reflects in the $RP_1@5$ value. Lastly, the figure shows that there are still files that stay *incorrect*. Specifically, the largest files 48 and 49 were not successfully translated once. Similarly, the surprisingly failing files 36 and 2 can not be translated by one of the models. Which is an interesting finding for the general performance that one could expect when working with LLM based code translation. Specifically, it seems that using different models can improve performance, because not all models can successfully translate all files, which is also visible in Figure 9.2. However, there are tasks that are difficult for all models, so using a few good models will not necessarily lead to a successful translation for all files.

While these aspects are interesting, the general focus is on the robustness. For this, the next section presents the fluctuations of the baseline, which enables the definition of thresholds that will later be used to distinguish non-robust behavior and nondeterministic noise.

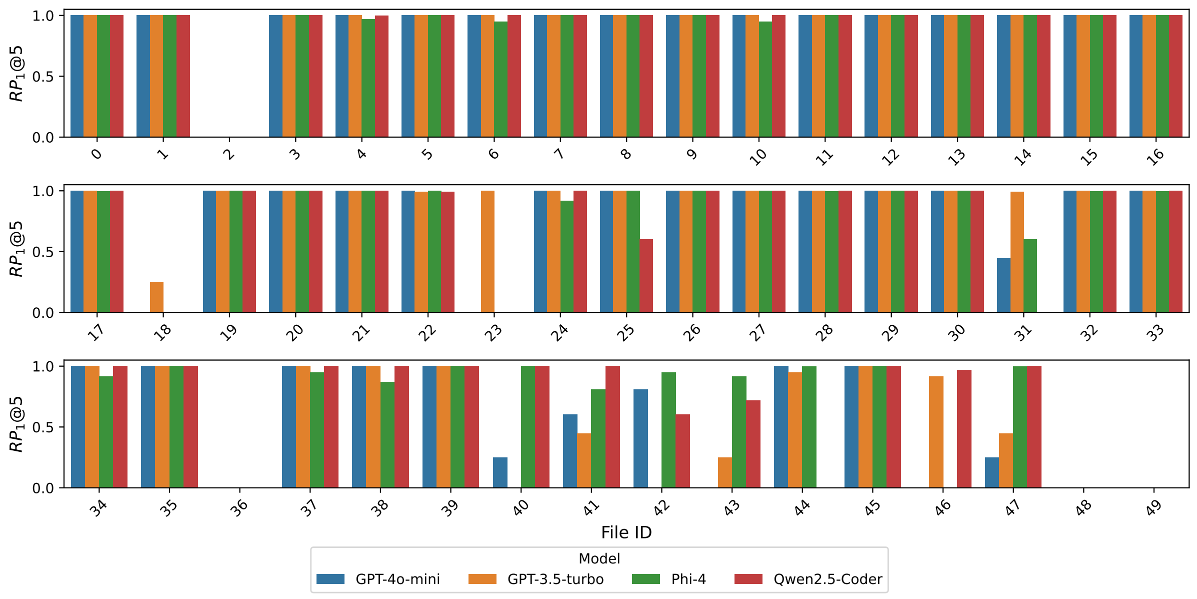


Figure 9.3: Comparing $RP_1@5$ at *fuzzing success* per file for all models.

9.2.3 Noise Analysis of the Different Models

To differentiate between expectable noise and non-robust behavior, the value distribution of $RP_1@5$ is being analyzed. Figure 9.4 illustrates the distinct model fluctuations. The plot details what could already be seen in Table 9.1, namely that *GPT-3.5-turbo* and *Phi-4* produce a higher *standard deviation*, which results in a larger distance between the *Z-Score* thresholds. Additionally, the distribution shows that the values for *GPT-3.5-turbo* and *Phi-4* are not symmetrically distributed. Both the form of the distribution and the high *standard deviation* negatively influence the trustworthiness of the *Z-Score* as a predictor for outliers.

For instance, the performance distribution for *GPT-3.5-turbo* is notably bimodal, with peaks occurring at approximately 0.8 and 0.87. Such bimodality produces a larger *standard deviation*, causing the calculated *Z-Scores* to extend significantly beyond the observed maximum and minimum $RP_1@5$ values. As a consequence, even deviations corresponding to *Z-Scores* lower than the threshold of 3.29 may signal non-robust behavior. Specifically, the distribution reveals that the maximum $RP_1@5$ value aligns with a *Z-Score* of 1.64, while the minimum is observed at a *Z-Score* of approximately 1.93. This observation highlights that, for *GPT-3.5-turbo*, the classical *Z-Score* threshold may overestimate the amount of fluctuations, potentially classifying non-robust behavior as expectable noise.

In contrast, *Phi-4* exhibits a negatively skewed distribution, which causes the high *standard deviation*. The distribution shows a high likelihood of achieving performance values near the maximum $RP_1@5$ 0.86, which has the *Z-Score* 0.72. Moreover, the negative *Z-Score* threshold of -3.29 corresponds to an $RP_1@5$ value of 0.72, yet the minimum performance is at 0.66 with an associated *Z-Score* of -5.18. This finding suggests that even minor improvements above the maximum ($Z = 0.72$) might be interpreted as non-robust, whereas performance below the typical negative threshold might still be normal noise, although with a relatively low probability. These discrepancies show that *Z-Scores* as predictors for outliers are heavily dependent on the underlying distribution characteristics, which seemingly vary significantly between models.

The implications of these findings are crucial when evaluating the models' robust-

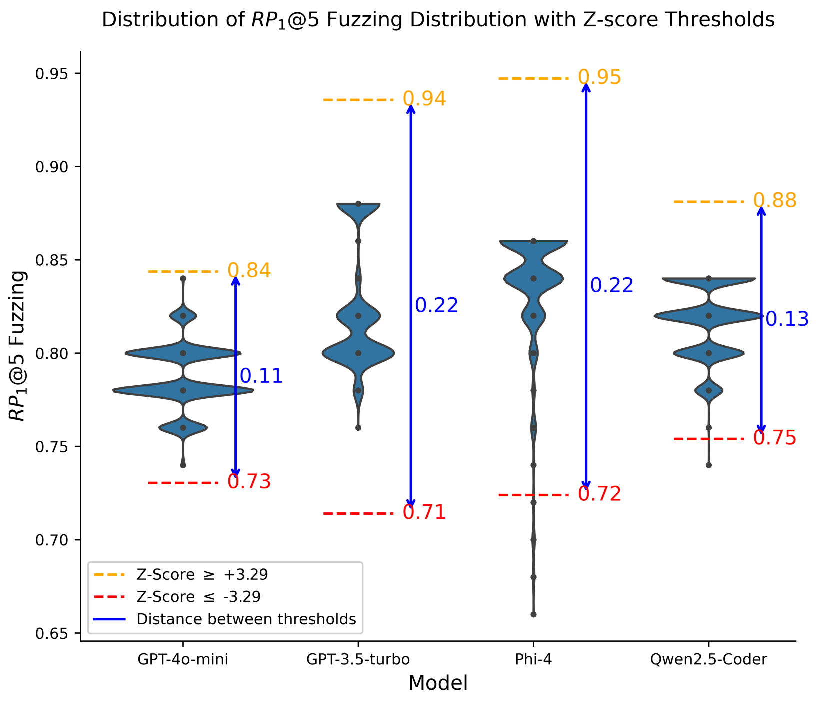


Figure 9.4: Violin plot [Ha98] showing the distribution of the $\binom{20}{5}$ RP_1@5 values for *fuzzing success* for the specific LLMs. In addition, the positive (orange) and negative (red) *Z-Score* thresholds and the distance between those (blue) are included.

ness under perturbations. They highlight that the application of the universal *Z-Score* threshold is not useful when the baseline performance distributions are asymmetric or multimodal. Instead, it is important to first assess the specific properties of each model's performance distribution before employing *Z-Scores* to distinguish between expected noise and meaningful non-robust behavior.

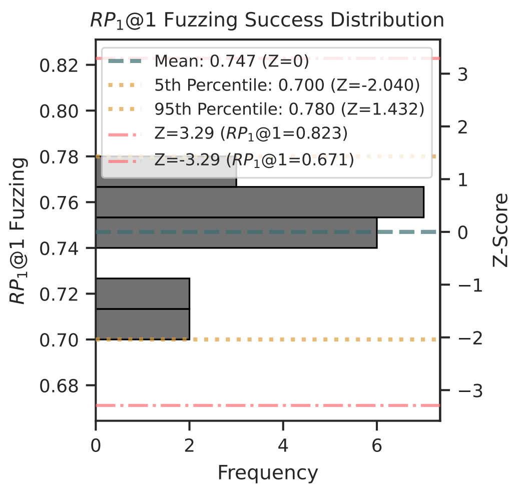


Figure 9.5: Distribution of *Qwen2.5-Coder's* mean RP_1@1 *fuzzing success* across the $\binom{20}{1}$ groups.

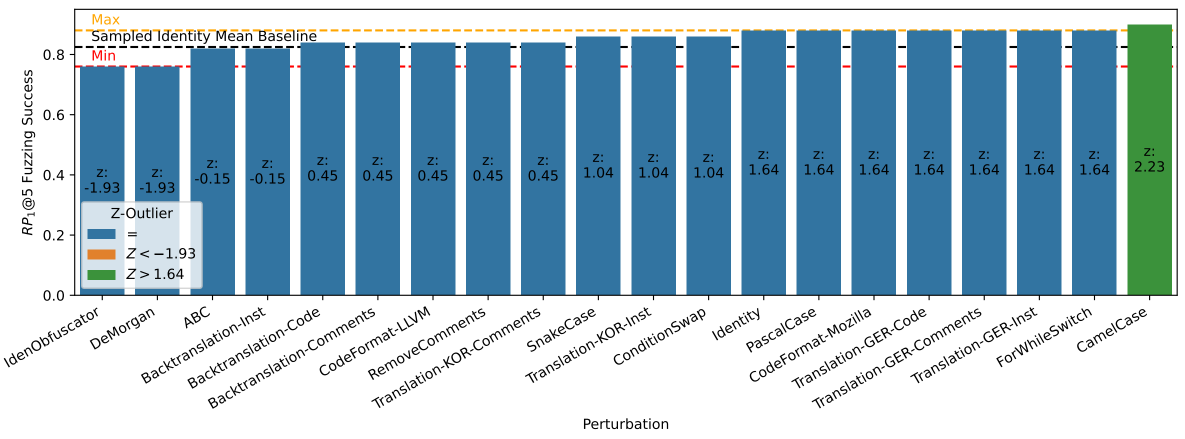


Figure 9.6: *GPT-3.5-turbo's fuzzing-success* under *deterministic perturbations*. The maximum and minimum are used to classify non-robust behavior.

As mentioned before, *Qwen2.5-Coder*'s baseline has to be analyzed for $n = 1$, $k = 1$ and is visualized in Figure 9.5. This figure demonstrates that even with only one run, the baseline performance is relatively good, which is also shown in Figure 9.1, where *Qwen2.5-Coder* proved to be the best model for one-shot experiments. The distance between lower and upper *Z-Score* threshold is 0.152 and according to the $\binom{20}{1}$ distribution slightly overestimates the fluctuations. Knowing that the robustness evaluation is conducted on a small sample size with only one run, such an overestimation might be desired to prevent false positive non-robustness claims. However, this should be kept in mind when evaluating the robustness.

9.3 Robustness under Deterministic Perturbations

With the information about the baseline performance for identical inputs, we can now investigate whether the *deterministic perturbations* cause significant outlying performance, which can be classified as non-robust behavior. The examinations for *GPT-4o-mini* revealed overall robust performance for the *deterministic perturbations*, with only a significant deviation under **IdenObfuscator** and **Translation-GER-Comments**.

The following examination investigates whether these findings generalize across different models and if the same perturbation strategies cause comparable robustness issues.

9.3.1 Robustness of GPT-3.5-turbo

Figure 9.6 demonstrates the robustness of *GPT-3.5-turbo* under *deterministic perturbations* with *feedback loops*. According to the 3.29 *Z-Score* rule, no perturbation caused significant performance changes. However, since the baseline distribution of *GPT-3.5-turbo* slightly overestimates the fluctuations, the maximum and minimum values are used as a reference. Most perturbations stay within this range, which prevents a clear classification of non-robust behavior

Recall that the baseline distribution of *GPT-3.5-turbo* shows a clear peak with a fast drop-off for higher values. This indicates that it is very unlikely to obtain performance scores above this peak. In this context, the **CamelCase** perturbation slightly exceeds the baseline's maximum, and is therefore considered non-robust. Although the performance

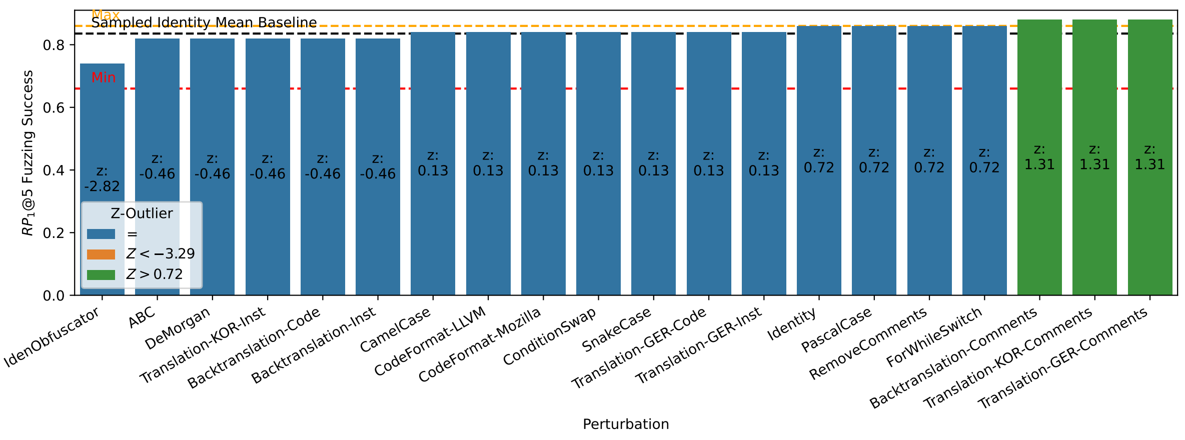


Figure 9.7: *Phi-4's fuzzing-success* under *deterministic perturbations*. The baseline's maximum and the negative *Z-Score* threshold −3.29 are used to classify non-robust behavior.

increase is only small, this rare event suggests a possible sensitivity of the model. Nevertheless, since the performance improvement is small, the confidence in it being a systematic sensitivity instead of a highly fortunate run is not very high.

Additionally, it is worth mentioning that **IdenObfuscator** produced the lowest performance, which is similar to the results observed for *GPT-4o-mini*. Although the drop in performance is not large enough to confidently classify it as non-robust, the consistent behavior suggests that this perturbation has characteristics that negatively affect model performance.

Overall, *GPT-3.5-turbo* shows a robust performance, with not a single *deterministic perturbation* causing really significant performance changes to the empirically examined baseline performance.

9.3.2 Robustness of Phi-4

Figure 9.7 presents the performance of *Phi-4* under *deterministic perturbations*. The baseline values of *Phi-4* showed a negatively skewed distribution with a clear peak and a fast drop-off, similar to *GPT-3.5-turbo*. Given this, performance exceeding the baseline maximum is very unlikely. Nonetheless, *Phi-4* achieves performance values beyond this maximum for the paraphrasing perturbations of comments (i.e., **Backtranslation**, **Translation-KOR**, and **Translation-GER**). Although the improvement is very small, the fact that all three perturbations produce such an effect suggests that *Phi-4* is sensitive to the natural language parts of code and may suggest that varying code explanations can improve its performance.

In addition, **IdenObfuscator** shows noticeably lower performance. Although neither the *Z-Score* nor the baseline's minimum indicates clear non-robust behavior, previous experiments support that **IdenObfuscator** represents a challenging perturbation for LLMs overall.

In general, *Phi-4* demonstrates primarily robust behavior. It only exhibits slight sensitivity to perturbations in code comments and confirms the difficulty posed by **IdenObfuscator**.

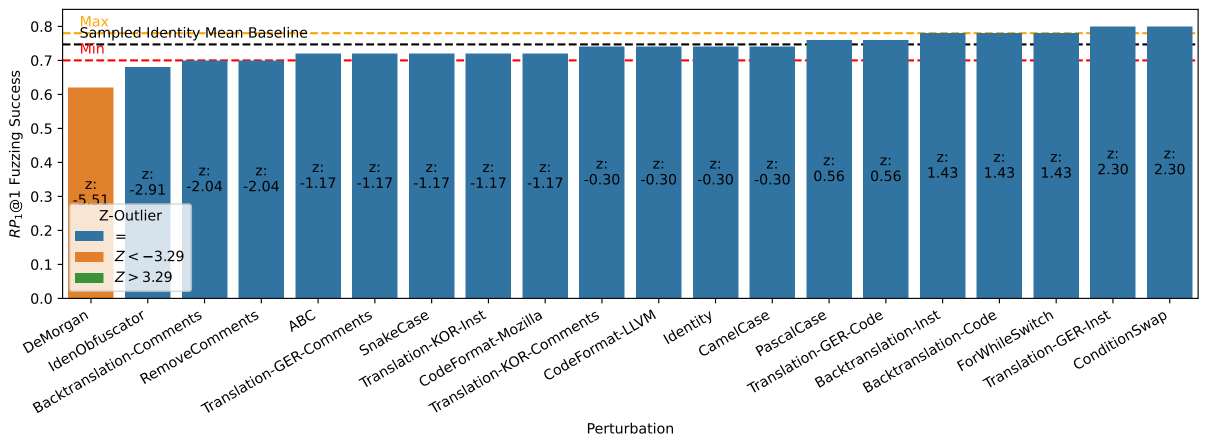


Figure 9.8: *Qwen2.5-Coder's fuzzing-success* under *deterministic perturbations* in one run $n = 1, k = 1$. The common *Z-Score* threshold 3.29 is used to classify non-robust behavior.

9.3.3 Robustness of Qwen2.5-Coder

Figure 9.8 presents *Qwen-Coder's fuzzing success* under one run with *deterministic perturbations*. Again, one can observe a robust behavior. According to the *Z-Score* threshold, only **DeMorgan** caused a significantly outlying performance to the distribution. Remember that *QwenCoder's Z-Score* distance also exceeded the maximum and minimum values of the distribution. Considering the maximum and minimum values as reference, would additionally show **Translation-GER-Inst** and **IdenObfuscator** as outliers. As the amount of performance increase is very small for the German instructions and the results are only based on one single run, it remains ambiguous whether *Qwen2.5-Coder* is sensitive to this perturbation, or if this is only a fortunate run. However, **IdenObfuscator** proves again to cause a meaningful performance drop, adhering to the previous findings.

DeMorgan consistently exhibited slightly lower performance compared to the baseline of the other models, although it did not fall below the robustness thresholds. Therefore, it remained unclear whether this difference was noise or an indication of a genuine robustness issue. *Qwen2.5-Coder* is the only model where this perturbation led to a highly significant performance drop. Since it is also the only model finetuned for code generation, this result may suggest that the model struggles with the unconventional logical constructs introduced by **DeMorgan**. However, as this observation is only based on a single run, an increase of n and k might reveal a different robustness profile.

Overall, *Qwen2.5-Coder* also demonstrates robust behavior, showing only a slight weakness with the problematic **IdenObfuscator** and a higher sensitivity to **DeMorgan** perturbations.

9.3.4 Perturbation Robustness for Different Models

The detailed analysis indicates that all of the models are mostly robust against *deterministic perturbations*. However, there are observable differences among the models. While *GPT-3.5-turbo* exhibits increased sensitivity to the use of **CamelCase**, both *GPT-4o-mini* and *Phi-4* appear to positively deviate from perturbations in the comments. Notably, all models experience a strong performance drop under the **IdenObfuscator**, suggesting that this specific perturbation poses substantial challenges when translating code from

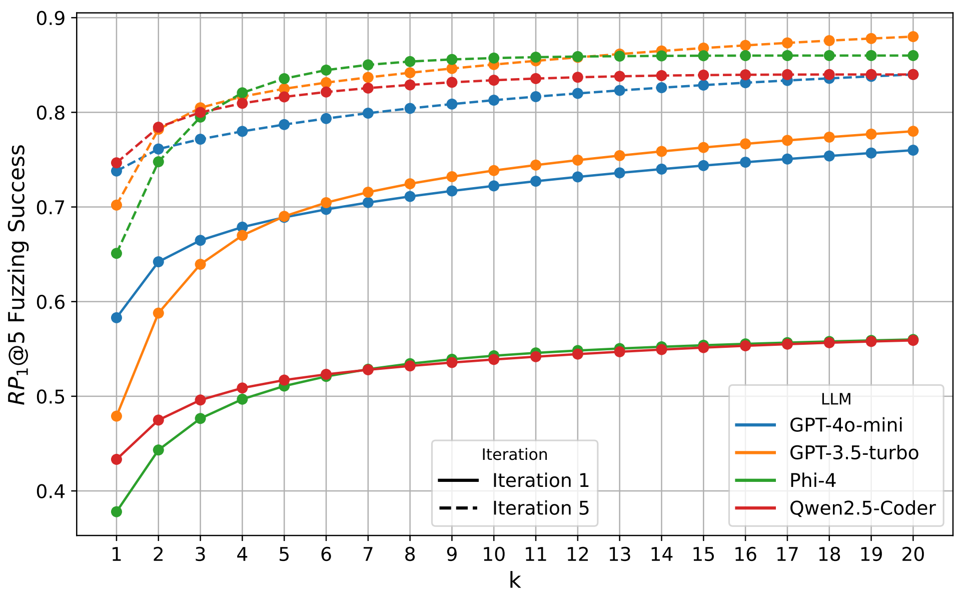


Figure 9.9: $RP_1@k$ for different k values of all 20 **Identity** runs with and without *feedback loops*.

C to *Rust*.

In summary, even though individual perturbations can have varying effects on each model, the overall level of observed robustness remains quite similar across the models. However, for this claim, it is also important to consider the range of output fluctuations. *GPT-4o-mini* showed the smallest fluctuations, which also reflects in the strictest non-robustness classification, implying that its practical robustness may be superior to that of the other models. Their larger variability observed in practice does not necessarily reflect a lack of robustness, rather, it might indicate a higher tendency for generating significantly different solutions for identical inputs.

The upcoming evaluation of the *feedback loops* will further clarify whether this general robustness is predominantly a result of the *feedback loop* approach, or if all models share a comparable inherent robustness when faced with perturbed inputs.

9.4 Evaluating Feedback Loops across Models

Before evaluating whether *feedback loops* consistently improve robustness for perturbations that primarily cause compilation issues (as observed for *GPT-4o-mini*), it is necessary to compare their overall impact across the different LLMs.

Figure 9.9 shows the *fuzzing success* for all models with $n = 20$ and increasing values of k, both with and without *feedback loops*. Overall, *feedback loops* lead to performance improvements across all models, but the increase is especially obvious for the local open-source models. Without *feedback loops*, these models are significantly less likely to succeed compared to the *OpenAI* models.

For *Phi-4*, previous results have shown that it responds well to changes in code comments, achieving better performance when comments are paraphrased. Moreover, *Phi-4* also benefits strongly from the *feedback loops* that provide compiler, linting, or fuzzing outputs. Since these outputs may act similarly to descriptive comments, the combined findings suggest that *Phi-4*'s performance improves when the input includes clear, well-formulated

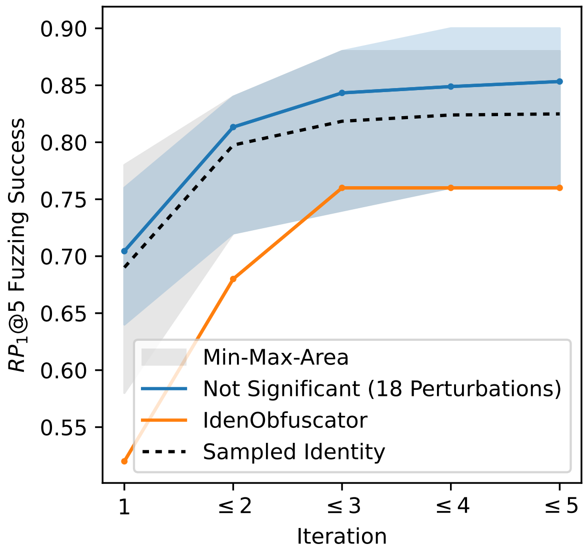


Figure 9.10: Impact of *feedback loops* on *GPT-3.5-turbo* on the only non-robust strategy of iteration one, classified by the maximum and minimum baseline values.

descriptions.

Furthermore, because the newly evaluated models perform no better than *GPT-4o-mini* at the relevant $k = 5$, they may also exhibit greater deviations from robust behavior when *feedback loops* are not applied.

9.4.1 Feedback Loops on GPT-3.5-turbo's Robustness

The impact of *feedback loops* on the robustness of *GPT-3.5-turbo* is visualized in Figure 9.10. The plot is similar to those presented for *GPT-4o-mini*, however, instead of classifying the perturbation in iteration one with the *Z-Score*, it uses the distribution's maximum and minimum values, as they do not overestimate the fluctuation for the model.

It can be observed how the *feedback loops* improve the general performance and also elevate the non-robust **IdenObfuscator** to the minimum value of the distribution. However, it is also visible that the fluctuations for the perturbations increase with higher iterations, leading to a previously not significant perturbation exceeding the maximum distribution, i.e. **CamelCase**. Additionally, it can be seen that not all perturbations get improved with the same amount, leading to at least one non-significant perturbation producing results similar to the minimum performance of the baseline. Specifically, in the last iteration of the *feedback loops*, this happens to **DeMorgan**. However, since it does not clearly produce worse results than the expected minimum, this does not necessarily mean it is an effect of the perturbation and could just be a normal fluctuation.

9.4.2 Feedback Loops on Phi-4's Robustness

Recall that *Phi-4* showed non-robust behavior for three comment paraphrasing perturbations with *feedback loops*. Figure 9.11 demonstrates that in iteration one, there were sixteen perturbations causing a significant baseline exceeding performance. As increasing iterations caused the elevation of the baseline, most of the initially outlying perturbations present normal performance deviations in higher iterations. In addition, since **IdenObfuscator** was not a significant outlier, it is represented in the blue area. We can observe the

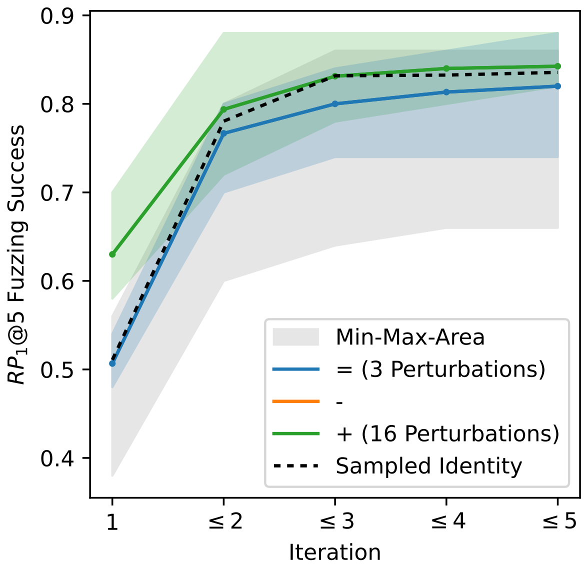


Figure 9.11: Impact of *feedback loops* on *Phi-4* on non-robust strategies of iteration one, classified by the maximum and minimum baseline values.

same behavior that could be seen for *GPT-4o-mini* and *GPT-3.5-turbo*, namely that the perturbation stops improving after two iterations, and therefore, based on the baseline, is classified as a non-robust perturbation.

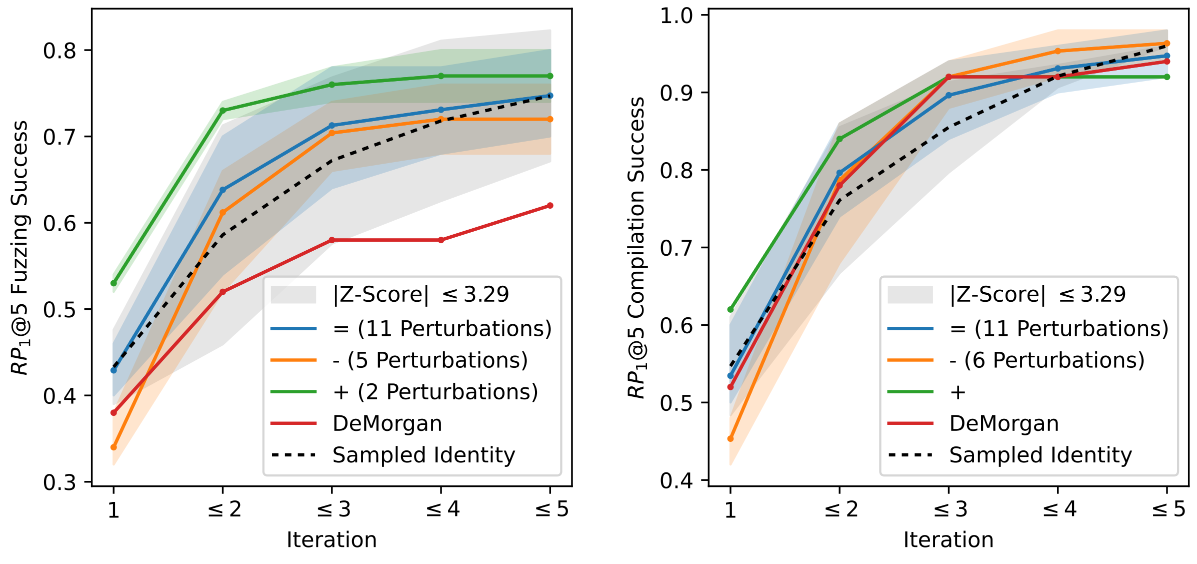
So overall, the *feedback loops* again reduce the magnitude of non-robustness, while this time the non-robustness shows by increased performance.

9.4.3 Feedback Loops on Qwen2.5-Coders's Robustness

Qwen2.5-Coder Figure 9.12a displays a lot of information. When only focusing on the first iteration, the figure shows eight outlying perturbations. Since **DeMorgan** remained the only *Z-Score* significant perturbation after all iterations, it was decided to present it as a single line to improve interpretability. With increasing iterations, all other outliers move closer to the baseline and therefore adhere to previous findings. With all *feedback loops* only **DeMorgan** out of the previously outlying perturbations, exceeds the *Z-Score* area, showing an almost perfect example for *feedback loops* increasing robustness.

Furthermore, as this is the only model that shows other negatively non-robust perturbations than **IdenObfuscator** in iteration one, it is meaningful to show the *compilation success*, to enable the evaluation whether the improvements are mostly due to easily fixable compiler and linting errors, or whether the *feedback loops* also fixed logical problems shown in fuzzing counterexamples.

Comparing outlying perturbations in *compilation success* in Figure 9.12b shows that multiple perturbations also caused significantly worse success for *compilation success*. After one *feedback* iteration, all of those perturbations are corrected to normal behavior, which similarly shows in Figure 9.12a. In addition, the loops elevated **DeMorgan** *compilation success* to a quite high performance by iteration ≤ 3. However, although the *compilation success* could be elevated significantly, the *fuzzing success* from iteration ≤ 3 to ≤ 4 did not increase. Only after a last increase of *compilation success* in ≤ 5, the *fuzzing success* increases. That interplay suggests that the most amount of increased *fuzzing success* is caused by *feedback loops* making more files compilable, which then could directly yield



(a) Fuzzing Success **(b)** Compilation Success

Figure 9.12: Impact of *feedback loops* on *Qwen2.5-Coder* on non-robust strategies of iteration one, classified by the 3.29 *Z-Score* threshold.

fuzzing success without iterations of fixing counterexamples.

These findings support the hypothesis that *feedback loops* increase robustness, especially when it is caused by failing compilations. The next section summarizes the information across the different models, underlining the model-agnostic consistency of the *feedback loops* impacts.

9.4.4 Overall Impact of Feedback Loops

Evaluating the models' robustness without the incorporation of *feedback loops* makes it clear that the observed robustness with *feedback loops* is strongly enhanced by the *feedback* iterations.

In the first iteration, each model exhibits different sensitivities to the applied perturbations. While both *GPT-4o-mini* and *GPT-3.5-turbo* perform similarly whether or not feedback is employed, the open-source models demonstrate significant deviations in their initial iteration, which are then effectively mitigated through the application of the *feedback loops*.

That goes in hand with Figure 9.9, which showed that the local models were significantly improved by the *feedback loops* and only with the help of those, presented comparable results to the *OpenAI* models. Consequently, applying a *feedback loop* strategy is always beneficial, not only for the general performance, but also for increasing the robustness.

However, despite the strong positive impact on performance and robustness, the current *feedback loop* strategy reaches its limits when addressing robustness deficiencies primarily caused by fuzzing counterexamples. An analysis combining the failures across all models, as presented in Figure 9.13, indicates that the strategy is predominantly successful in solving issues related to the *compiler* and *clippy* checks. Although some failures caused by fuzzing are resolved, the majority of these errors remain uncorrected even after five iterations. Moreover, as noted previously, simply increasing the number of iterations does not necessarily improve the performance in the same way. Instead, a saturation effect in

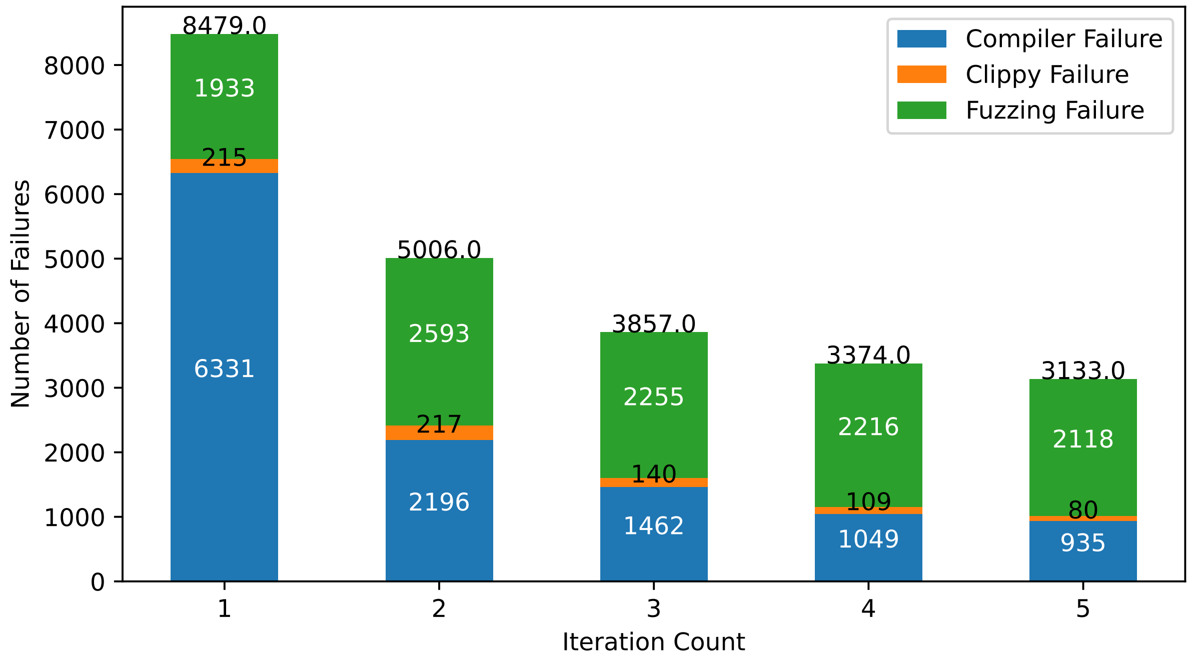


Figure 9.13: Detailed analysis of which check fails the translation task for each iteration of the feedback loop for *all models* under *deterministic perturbations*.

the *feedback loop* approach could be observed after two to three iterations.

Hence, for future work, it may be beneficial to try multiple variances of *feedback loops*. For instance, implementing a strategy that starts an entirely new attempt after three iterations fail, in order to maybe receive a better result due to fluctuation. Similarly, one could combine and entangle multiple models for each iteration, as the *Venn diagram* showed that different models have different success in solving different files. Before going into a detailed discussion of all findings, it is necessary to show whether we can observe differing correlations between similarity and robustness for different models.

9.5 Semantic Similarity in a Cross-Model Context

Chapter 8 came to the conclusion that for *GPT-4o-mini* and the *ada-002* embedding model, there is no clear correlation between perturbation similarity and translation success. While the correlation is slightly higher without *feedback loops*, it vanishes when applying *feedback loops*.

This section provides information, whether this finding also applies to other LLMs. The assessment utilizes the same embedding vectors and shows the correlation between the models' translation success. For simplicity, the focus of this analysis is only on the perturbations on the code part, as these are the relevant perturbations for the code translation system.

9.5.1 GPT-3.5 and Semantic Similarity

Figure 9.14 demonstrates a similar behavior to what could be seen for *GPT-4o-mini*. The least similar perturbation **ABC** causes a robust translation success with and without *feedback loops*. Similarly, with *feedback loops*, the **DeMorgan** perturbation shows an almost

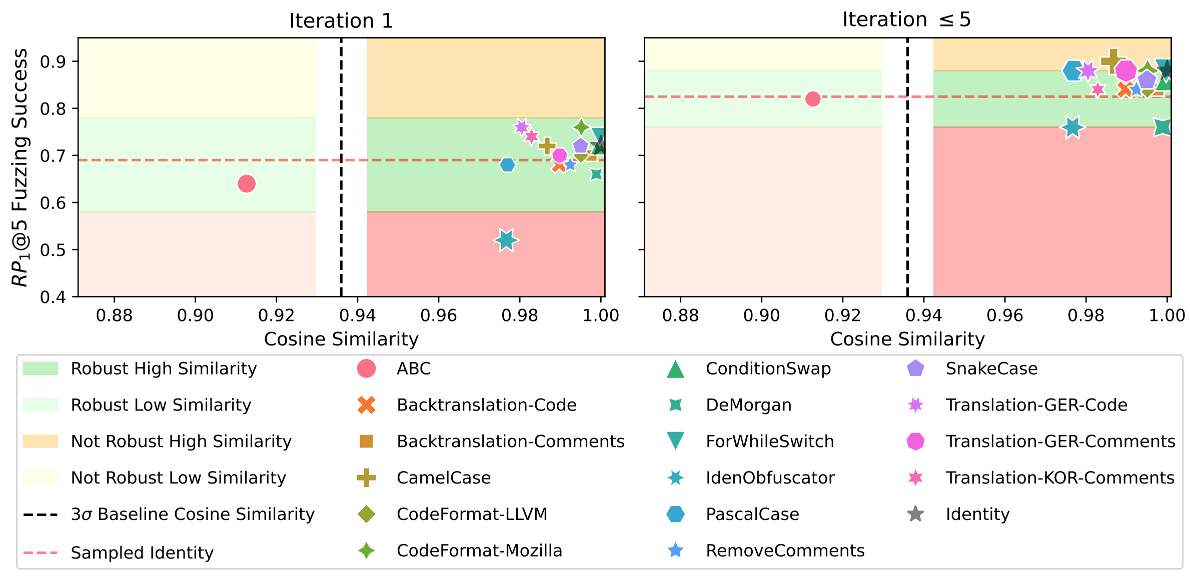


Figure 9.14: *GPT-3.5-turbo's* correlation between *cosine similarity* and RP_1@5 with and without *feedback loops*. The minimum and maximum range of the baseline is used to define the robust area.

significant drop, while being highly similar. Furthermore, it seems that perturbations with very high similarity scores show less deviation from the baseline than those with less similarity.

Calculating the *Pearson correlation coefficient* [SBS18] between RC@P suggests a weak correlation without *feedback loops* (−0.226). Similarly, the correlation coefficient on the results with *feedback loops* also shows weak correlation(0.279). However, since it is not negative, it suggests that perturbations with higher similarity scores yield less translation success. **ABC** has a strong impact on this coefficient, as it has an outlying similarity, subsequently with it yielding less change with *feedback loops*, the correlation is flipped.

Consequently, for the other *OpenAI* model, there is also no clear correlation that could be used as a predictor for translation robustness.

9.5.2 Phi-4 and Semantic Similarity

Figure 9.15 details the high amount of perturbations that exceeded the baseline's maximum performance in iteration one. It might suggest that there could have been erroneous translations during the baseline run, but manual investigation showed that the framework worked as expected. Knowing the data points are valid, we can again observe that very high similarity scores produce similar translation success, and with *cosine similarity* values below ≈ 0.995, the perturbations start to deviate. With the *feedback loops*, this gets less noticeable, which again also shows in the *Pearson correlation*: -0.237 (weak without *feedback loops*) and -0.097 (negligible after five iterations).

9.5.3 Qwen2.5-Coder and Semantic Similarity

For *Qwen2.5-Coder* and $k = 1$, there seems to be even less correlation (see Figure 9.16). The data points do not follow any pattern and also deviate for very high *cosine similarity* values. Although the very similar perturbations **ForWhileSwitch** and **ConditionSwap** only

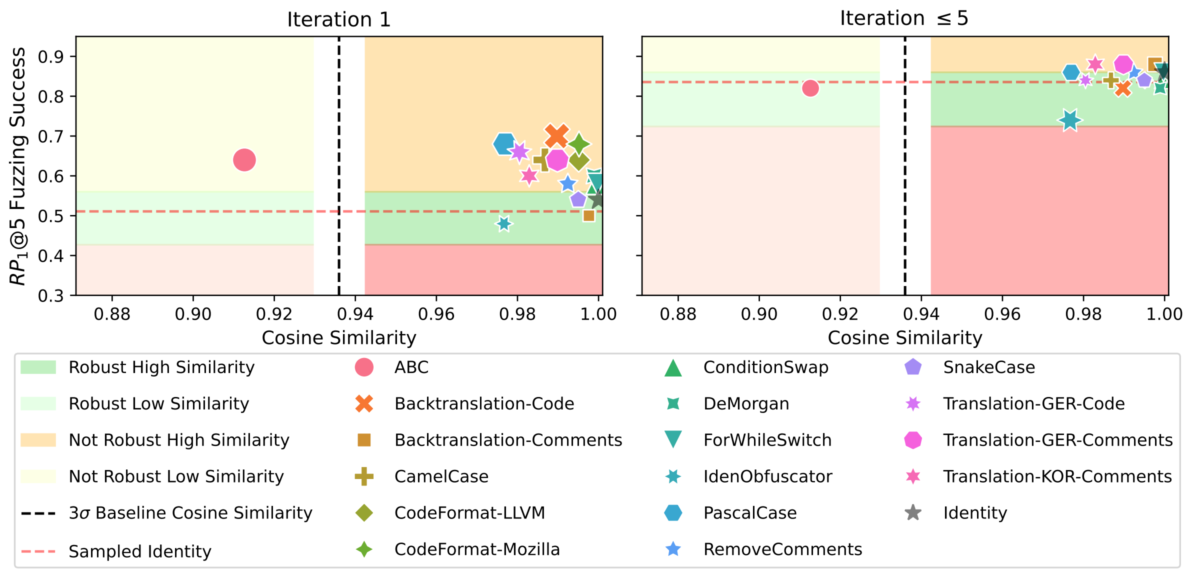


Figure 9.15: *Phi-4's* correlation between *cosine similarity* and RP_1@5 with and without *feedback loops*. The *Z-Score* of −3.29 and the maximum range of the baseline are used to define the robust area.

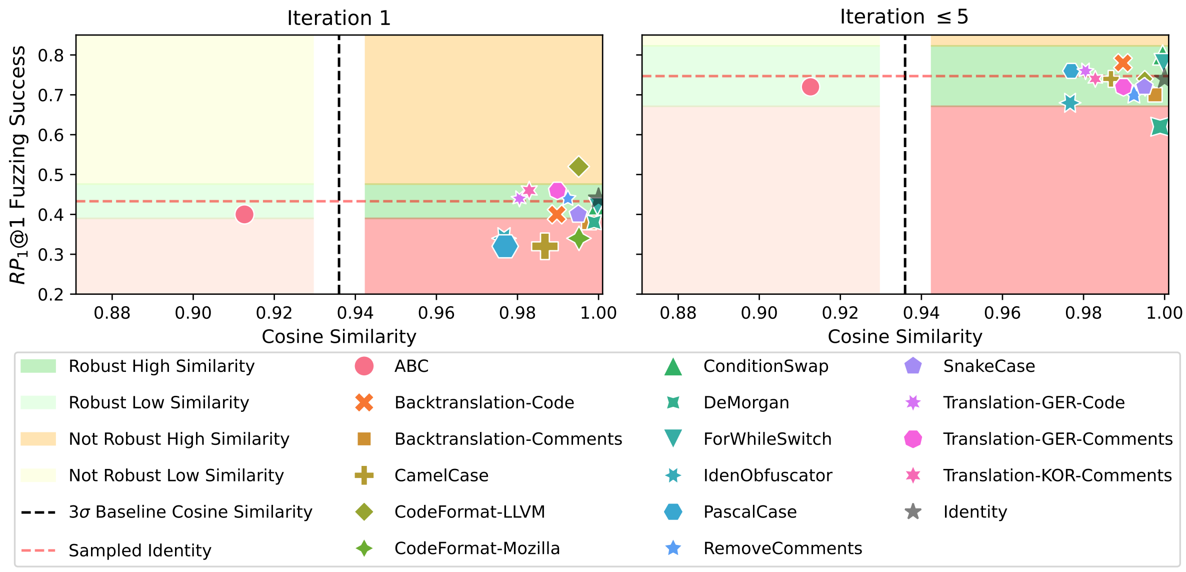


Figure 9.16: *Qwen2.5-Coder's* correlation between *cosine similarity* and RP_1@1 with and without *feedback loops*. The *Z-Score* of ±3.29 of the baseline is used to define the robust area.

slightly deviate from the baseline without *feedback loops*. Other highly similar perturbations like **DeMorgan** or **Backtranslation-Comments** show outlying performance in iteration one. As detailed before, the **DeMorgan** perturbation causes the strongest performance drop with *feedback loops*, although it is one of the perturbations with the highest similarity scores. The *Pearson correlation coefficient* confirms what was observed visually. Without *feedback loops* there is a negligible correlation (-0.034), and with *feedback loops* a weak correlation (0.183).

Combining these findings highlights again that there is no linear correlation between *cosine similarity* and translation success.

9.5.4 Overall Correlation between Similarity and Translation Success

The model-specific examination reinforced the findings that could be observed for *GPT-4o-mini*. Without *feedback loops*, there is very little correlation, which vanishes entirely when *feedback loops* are applied. Consequently, using *cosine similarity* to predict the robustness of certain perturbation strategies appears ineffective. At least this is the result for the proposed perturbation strategies and with the embedding model *ada-002*. It remains possible that alternative, more code-specific embedding models might capture similarity in different ways, leading to other correlation outcomes. Furthermore, since the perturbations were designed to generate semantically similar code, applying additional perturbation strategies that introduce more substantial modifications to the original files might reveal the expected correlation.

Given that this aspect is not the primary focus of the thesis, the conclusion remains that the translation success for similar inputs is influenced by highly individual factors or pivotal tokens, making it challenging to predict whether a particular perturbation will cause robustness deficiencies.

9.6 Interpretation for RQ5

The examinations of this chapter investigated whether the previous conclusions about robustness with *GPT-4o-mini* can be transferred to different LLMs. With the knowledge and conclusions of the particular comparisons, we can now empirically discuss RQ5: “Are robustness results consistent using different LLMs?”

As demonstrated in Section 9.2, the various models exhibit notably different baseline performances. While it was expected that different models would have different probabilities of translating files successfully, it was surprising to observe that the general distribution of those probabilities is drastically different for each model. This led to the conclusion that always classifying non-robust behavior with the *Z-Score* is not effective. It rather demonstrated that a thorough investigation of the baseline performance is always necessary to interpret what performance deviations reflect non-robustness and what may be caused by the nondeterministic nature of LLMs. This is also an interesting finding for RQ1 and will be revisited in the concluding discussion of the thesis in Chapter 10.

The evaluation of robustness under *deterministic perturbations* revealed substantial differences among the models when *feedback loops* were not applied. While the **IdenObfuscator** represented a single perturbation that resulted in significant performance drops across all models, other perturbation strategies caused varying responses. Incorporating *feedback loops* always led to dramatic increases in both performance and robustness, demonstrating that such loops are generally superior and should be incorporated in practical applications.

Nevertheless, even with *feedback loops*, every model exhibited at least one perturbation that stood out as a significant robustness outlier. In addition, the discovery that *feedback loops* predominantly solve issues with *compiler* or *linter* errors could also be transferred across models.

By aggregating the mean performance ($\boldsymbol{RP}_1@k$) and the robustness relative to the baseline ($\boldsymbol{RC}_1@k$), one can interpret and compare the usability of the different models.

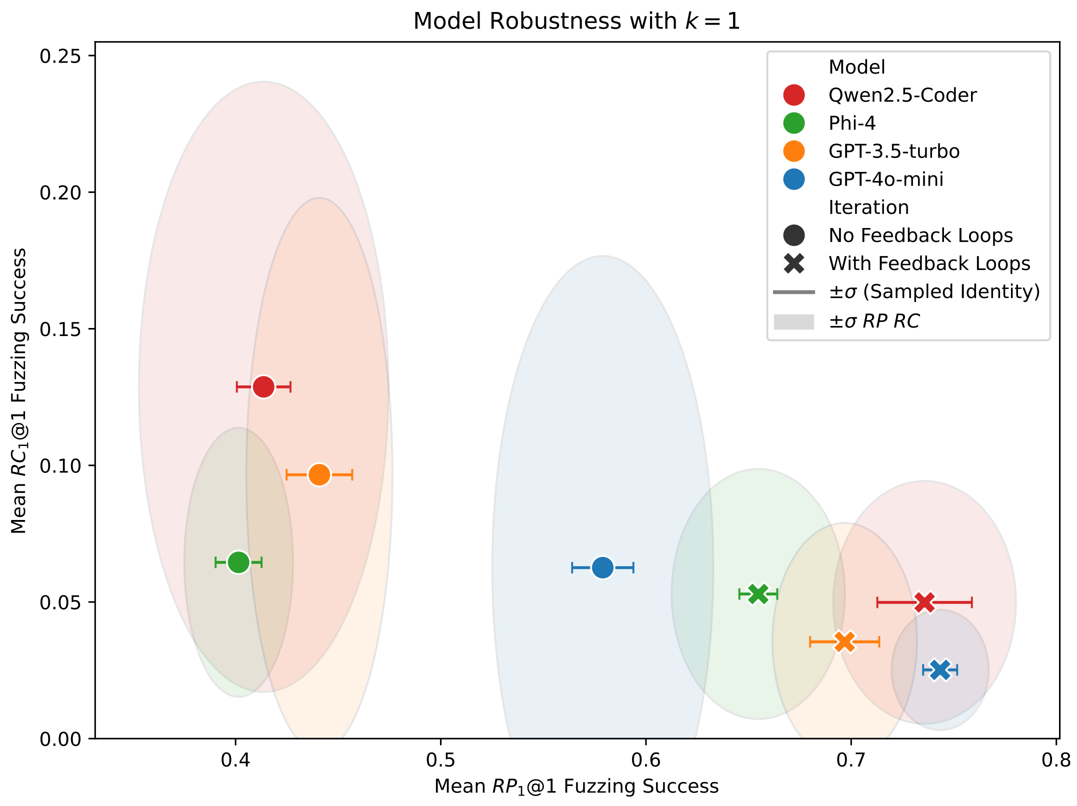
Figure 9.17 supports what has been detailed earlier. The optimal choice of model can depend on the number of runs one is willing to invest. When considering a single run ($k = 1$), *GPT-4o-mini* demonstrates superior performance and exhibits the smallest deviation from its baseline under perturbations. While this difference is highly noticeable with *feedback loops*, it still remains after all iterations of the feedback. This finding somewhat aligns with the model comparison over different k in Figure 9.9, where *Qwen2.5-Coder* exhibited the best one-shot performance, closely followed by *GPT-4o-mini*. Under perturbations, one can observe that *Qwen2.5-Coder* yields more performance deviations under perturbations than the *OpenAI* models.

When the evaluation is expanded to five runs, as shown in Figure 9.17b, the initial superiority of *GPT-4o-mini* dissolves once feedback loops are incorporated. In this context, *GPT-3.5-turbo* yields higher translation success but with the drawback of reduced robustness, whereas *Phi-4* shows slightly lower translation success yet exhibits greater insensitivity to perturbations. Additionally, the standard deviation derived from the *Sampled Identity* reveals that *GPT-3.5-turbo* and *Phi-4* yield less consistent results for identical inputs. Therefore, the choice of model must be tailored to the specific requirements of the application of the code translation system.

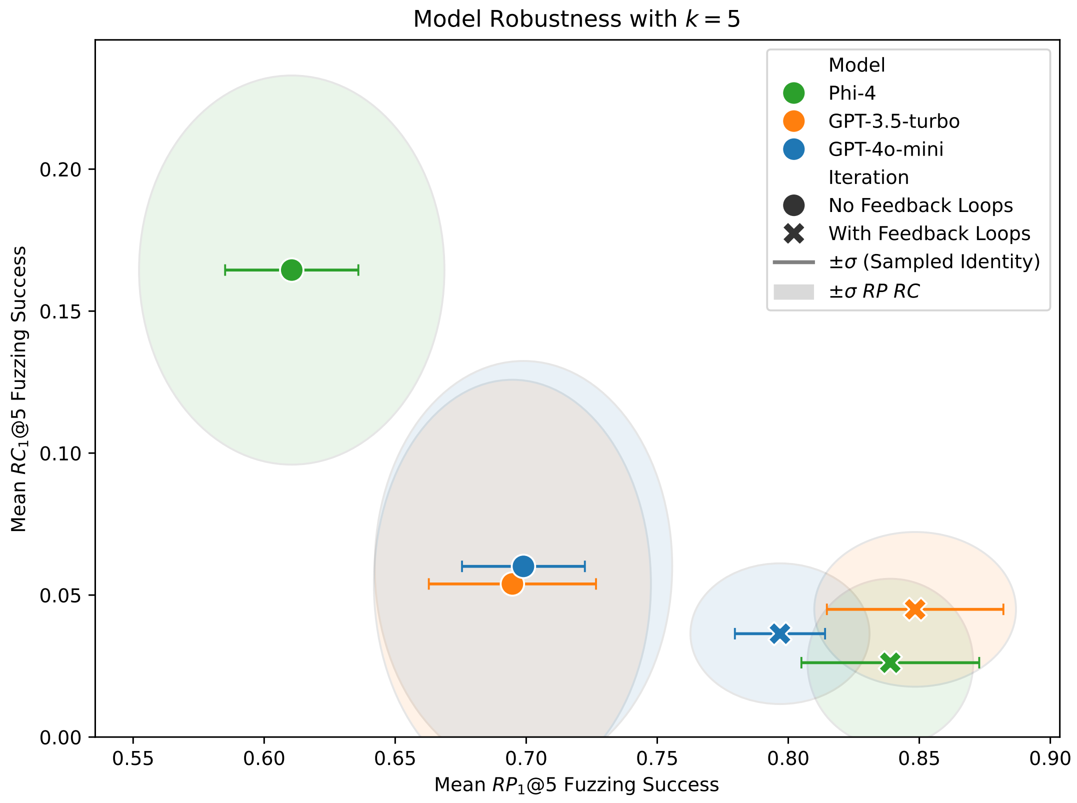
Recall that the robustness analysis of *GPT-4o-mini* in Chapter 6 also incorporated the error rates of the system. At this point, the error rates for the other models have not been discussed. Appendix A.4 displays the different error rates per perturbation for each model. It was observed that the additionally examined models *GPT-3.5-turbo*, *Phi-4*, as well as *Qwen2.5-Coder* resulted in slightly higher error rates than *GPT-4o-mini*. While this is a noteworthy finding, it does not impact prior conclusions about robustness, because the error rate for the baseline **Identity** was comparably elevated. Also, with *GPT-4o-mini's* low error rate and *GPT-3.5-turbo* successfully translating the highest amount of files (Figure 9.2), these errors are most likely a result of incorrect translations, produced by the models themselves. As mentioned, Figure 9.9 showed that *GPT-4o-mini* had a significantly higher success rate on $k = 1$ without *feedback loops* compared to the other models. However, when applying the *feedback loops* with two additional attempts ($k = 3$), the figure shows that *GPT-4o-mini* performs worse than all other models. So the errors are likely not due to the experimental setup or implementation, but rather due to a genuine weakness of the models. Since all errors are also indirectly incorporated in Figure 9.17, this only reinforces the need for a requirement-specific choice of the LLM.

Imagine a scenario where cost and hardware limitations are no factor and the setup automatically performs five runs. In this context, in experimental setups, where the goal might be the highest translation success, *GPT-3.5-turbo* is suggested to be the best fit. In environments that pose challenges through highly inconsistent code or badly written code, the choice could be to translate with *Phi-4*. However, when the requirement is to produce consistent results for identical inputs, with light deficits in robustness and translation success, the decision could be to use *GPT-4o-mini*.

This digression shows that although the models are quite robust and show similar behavior on *feedback loops*, even small nuances can reveal strengths and weaknesses of



(a) $k = 1$



(b) $k = 5$

Figure 9.17: Comparison of performance and robustness in *fuzzing success* for the evaluated LLMs under *deterministic perturbations*. The ellipses represent the standard deviations of the aggregated metrics $RP_1@k$ and $RC_1@k$, highlighting the variability among measurements. Additionally, the standard deviation of the *Sampled Identity* is included to reflect the baseline's noise due to nondeterministic fluctuations with identical inputs.

different LLMs.

It is also noteworthy that the observation of *cosine similarity* not strongly correlating with translation success maintains true for all the models examined.

Combining this information for RQ5 suggests that high-level robustness results are consistent among models, whereas the effects of single perturbations can differ significantly. Nonetheless, the consistent impact of **IdenObfuscator** suggests that some perturbations pose inherent challenges when LLMs are tasked with translating *C* code into *Rust*.

10 Discussion

This chapter summarizes the robustness evaluation results found by applying the proposed framework on SOTA LLMs in prior chapters. The information is used to give detailed and compact answers to the RQs that arose due to the significant research gaps in this area (Section 1.2). Furthermore, the chapter discusses threats to the validity of these findings and presents improvements that can be addressed in future work. Lastly, Section 10.5 elaborates on the implications of the results of the thesis and whether the LLM-based code translation system is Sáenz's *"benefactor"* or rather a *"veritable public enemy"*.

10.1 Summary and Contributions

This thesis presented a comprehensive framework for a systematic robustness evaluation of LLM-based *C-to-Rust* code translation that addresses the need for code-specific perturbations, LLMs-agnostic evaluations, as well as nuanced aspects like *feedback loops*, *semantic similarity*, and noise differentiation. The main contribution is a three-step framework (Figure 4.1) that enables the assessment of how LLMs handle realistic variations in the input for the use case of code translation.

Step I of the framework performed perturbations, i.e., various strategies to generate semantically similar, but structurally different variations of an original prompt containing a natural language instruction and the *C* code that is to be translated, which the thesis named **Identity**. A total of 23 distinct perturbation strategies were applied, targeting either the instruction, comments, or code across six levels of complexity described in Section 4.2.2, and mentioned in Table 4.1. This directly met the need for code-focused input variations that go beyond instruction paraphrases. The perturbed prompt and the **Identity** were systematically fed into an existing SOTA *C-to-Rust* translation system in *Step II* [QHW25], using selected LLMs and a *generate-and-check* pattern to directly verify translation success. In detail, the system incorporated checks for *compilation success* and *fuzzing success* that verified for *functional equivalence* with *differential fuzzing* (Section 4.3.4) and applied *feedback loops* in case of checking failures. This specific setup allowed for the robustness investigation of the identified research gaps and robustness nuances absent in the literature regarding an assessment specifically for code translation, languages *C* and *Rust*, as well as the behavior of modern LLMs and the impacts of *feedback loops*. *Step III* enabled the multi-dimensional analysis of the translation outcomes. It employed established (RP_s@k, adapted from Wang et al. [Wan+23]) and novel (RC_s@k) robustness metrics, complemented by a semantic similarity analysis using embeddings (Section 4.4), allowing a nuanced assessment whether semantic similarity correlates with robustness deficits.

An important part of the framework's application methodology (Section 4.5) involved establishing model-specific baseline distributions for repeated translations of unperturbed inputs. As detailed in Section 5.2 and Section 9.2, this allowed distinguishing between inherent LLM noise and actual robustness deficits caused by perturbations.

The framework was used with the 23 perturbation strategies. The evaluation consisted of 20 *deterministic* and 48 *stochastic* perturbed variations of the prompts (68 in total), when considering different perturbation targets, $s = 3$ variants for *stochastic* strategies, along with the **Identity** baseline. The evaluation used a curated 50-file *C* benchmark dataset primarily including industry-relevant internal automotive code (Section 5.1, Figure 5.1). The largest robustness analysis was performed for the modern and widely-used *GPT-4o-mini*, using all 68 prompt variations. This analysis started by aggregating results for different groups of perturbations and later examined perturbation-specific as well as file-specific effects in depth. In addition, the analysis compared results among different iterations of the *feedback loops*, and incorporated the semantic similarity in the form of *cosine similarity* into the evaluation. While the analysis had the highest detail for *GPT-4o-mini*, for additional and comparative insights, the thesis also investigated *GPT-3.5-turbo*, *Phi-4*, and *Qwen2.5-Coder-14B* on *deterministic* perturbations with a lower level of detail (Section 9.3).

The empirical data collected from the systematic investigation detailed in Chapters 5 to 9 form the basis for answering the RQs that arose due to the identified research gaps.

10.2 Answering the Research Questions

Based on the evaluation results of the framework, this section answers the RQs introduced in Section 1.3 and summarizes the findings from Chapters 5 to 9 to provide the necessary evidence.

10.2.1 RQ1: Components for a Comprehensive Robustness Evaluation

RQ1 is the main RQ for this thesis and presents the basis for all following and in-depth questions. It asks: "What methodologies and components should be integrated into a comprehensive evaluation framework to assess the robustness of an LLM-based code translation system?" By applying the framework on a benchmark dataset, this thesis empirically showed that a comprehensive robustness evaluation framework for LLM-based code translation requires several key components and methodological principles: *(i)* the use of systematic and diverse generation of perturbations that reflect real-word variations in code, *(ii)* a translation pipeline with automatic verification of functional equivalence, as well as *(iii)* multi-faceted performance metrics and a suited methodological application of the framework to employ the metrics and enable a systematic noise differentiation. Due to the empirical results, each of the components can be directly justified:

***(i)* Perturbations** As already shown by previous work, robustness should be examined by employing semantically similar, but structurally different input variations [Wan+23; Mas+23; Yan+23a; Imp+25]. However, the thesis reasoned that such input variations have to go beyond simple natural language instruction variations for the code translation use case, since code translation rather works with a fixed instruction in the prompt. To evaluate a code translation system for practical usage, it needs an assessment of whether the model behaves unexpectedly for prompts with semantically similar, but structurally varying source codes. Moreover, these variations should not be limited to superficial modifications that only focus on comments or identifier names, but also make profound changes to the code, to test the model's capabilities for scenarios with differently formatted or even equivalent implementations with differing control flows. For a better classification of a perturbation's "depth", the thesis used the six levels of code changes introduced by Faidhi and Robinson [FR87].

The results of the experiments proved that all these additional considerations were necessary. Specifically, just using instruction perturbations would have missed key code-variational weaknesses. Namely, most models suffered a consistent and significant drop in performance to the **IdenObfuscator** (*GPT-4o-mini* in Figure 6.1a, *GPT-3.5-turbo* in Figure 9.6, and *Qwen2.5-Coder-14B* in Figure 9.8) strategy, a robustness issue that would have been hidden without direct code perturbation. Another example is the *stochastic* **LLMCodeExtraction** perturbation that caused major performance drops for *GPT-4o-mini* and led to the conclusion that the LLM might show weaknesses for code with higher amounts of nested functions. Furthermore, the various code perturbations that did not cause significant deviations build confidence that varying casing styles or formatting styles will not *negatively* impact the translation performance of the evaluated, modern LLMs. Additionally, while it could not be shown that deeper perturbation levels necessarily cause stronger robustness deficits (Table 6.3 and Table 6.7), using the levels as orientation was still beneficial. It enabled a systematic exploration of perturbation strategies in general, and additionally revealed that sensitivities were strategy-specific, not complexity-dependent (e.g., **IdenObfuscator** in Level II consistently causing stronger performance deviations than the Level VI **ConditionDup**). Thus, it was illustrated by the thesis that LLMs can be robust to variations at all levels, even containing variations in decision logic, but can simultaneously be weak for particular strategies. So, a robustness evaluation must be comprehensive and cover a broad range of diverse perturbation strategies to reflect different scenarios and enable an assessment that does not overestimate the robustness. These mentioned findings could have gone unnoticed without these perturbation levels and also align with Nezhurina et al.'s [Nez+24] concerns regarding generalization. The experiments in the thesis demonstrated that while there is robustness against the high-level **ConditionDup** perturbation in Level VI, this does not imply that there is robustness against the lower-level **IdenObfuscator** in Level II. That clearly reveals that level-wise robustness is not transferable.

***(ii)* Translation Pipeline Verifying Functional Equivalence** To assess the robustness, it is necessary to quantify the performance of the system. Since the ultimate goal of a successful code translation is *functional equivalence*, verifying this characteristic is crucial to quantify a system's performance. Furthermore, to get SOTA translation performance, a code translation system uses the auto-repairing *feedback approach*, which, in addition to repairing functional equivalence counterexamples, also repairs translations that cannot be compiled or contain linting errors. Moreover, it is necessary to account for a model's nondeterminism by leveraging multiple translation attempts. The evaluation results demonstrate that considering these aspects is important for a comprehensive robustness evaluation.

Due to many translations passing compilation but failing functional equivalence checks Figure 5.5, relying on compilation alone is insufficient. Hence, it is necessary to test equivalence directly via differential fuzzing to measure robustness relevant to practical usability. In addition, Section 5.2, Section 9.2, and specifically Figure 9.4, highlight the necessity of repeated translation attempts. Due to the significant performance fluctuations observed even with the same input, it is necessary to run multiple times, requiring *pass@k*-based metrics due to this nondeterminism. Moreover, Chapter 7, or Figure 9.9 in particular, detailed the importance of integrating the *feedback loops*, as they drastically improved the overall translation performance and also robustness, which will be concluded in Section 10.2.3. Without *feedback loops*, the open models would have been judged significantly worse. Since *feedback loops* are highly beneficial, only with the drawback of an increased number of LLM calls, practical applications will most likely make use of such strategies. Consequently, assessing the robustness only without *feedback loops* would not have been as practically relevant for LLM-based code translation systems. However, ignoring the system's performance without *feedback loops* completely would have prevented the evaluation from revealing profound weaknesses that could only be due to the iterative repairs (Figure 7.6, Figure 9.12).

***(iii)* Performance Metrics and Application Methodology** The perturbations and the translation pipeline produced vast translation statistics. However, to comparably quantify robustness, it needs specific robustness metrics and strategies for applying the first two components in an accurate evaluation that distinguishes between genuine robustness deficits and nondeterministic model noise.

The established $RP_s@k$ metric by Wang et al. [Wan+23] was beneficial for examining translation performance for *stochastic perturbations* that can produce different variations for the same file, where one could lead to stronger deviations than another. $RP_s@k$ as a performance metric enabled quantifying the translation statistics for the entire dataset and over the repeated translation attempts. In addition, the introduction of $RC_s@k$, which measures the relative performance deviation to a baseline, enabled an intuitive judgment of the magnitude of perturbation impact. That was especially beneficial for RQ3 and evaluating the impact of *feedback loops* on robustness (Figure 7.7), or when comparing the robustness of different models addressing RQ5 in Figure 9.17.

Furthermore, the proposed aggregation strategies in Section 4.5.2 are a valuable component of the framework to reveal robustness results for perturbations across the entire dataset or by combining perturbations of different characteristics, such as their target or perturbation level. Nonetheless, when wanting to interpret and identify reasons for perturbation issues, results at the file level became helpful (Section 6.1.3 or Section 6.2.3).

Moreover, the statistical methodology to distinguish between noise and robustness deficits (Section 4.5) turned out to be indispensable. Figure 9.4 showed that different models can produce significantly different noise profiles, which completely prevents a comparative analysis, even when the LLMs are used with identical sampling strategies and configurations. Only comparing $RP_s@k$ and $RC_s@k$ across models and perturbations would have resulted in inconsistent conclusions about the effects of perturbations. Since this is the focus of RQ2, it will be discussed in more detail in Section 10.2.2. Similarly, including the semantic similarity of perturbations was beneficial to evaluate whether embedding-based similarity can be used as a prior predictor of robustness, which will be concluded in Section 10.2.4.

Conclusion of RQ1 In summary RQ1 can be answered as follows:

Answer to RQ1:

A comprehensive robustness evaluation for LLM-based code translation systems should involve the combination of *(i)* diverse, code-focused perturbations covering multiple complexity levels, *(ii)* a task-specific *generate-and-check*-based translation pipeline with *feedback loops* and a capability of performing repeated translation attempts, as well as *(iii)* an evaluation methodology with adequate metrics, systematic noise separation, fine-grained performance aggregations, and incorporating context like perturbation similarity.

10.2.2 RQ2: Distinguishing Between Model Noise and Robustness

RQ2 targets the model noise and asks: "How does one differentiate between inherent LLM nondeterminism (noise) and true robustness deficits?" As detailed earlier, LLMs are stochastic, nondeterministic models that can produce different outputs for identical inputs in repeated attempts. In the context of a robustness evaluation, which aims to quantify whether perturbations cause significant deviations in performance, this poses a challenge that has to be solved. An observed performance change could reveal a genuine lack of robustness or represent a normal fluctuation that has to be accepted when working with LLMs. If there is no proper strategy, it is possible that the human evaluator using the framework may misinterpret results and classify noise as non-robust behavior or vice versa. Section 4.5.1 presented a statistical baseline methodology to tackle this problem. The basic idea of this is to determine how much we expect LLM's performance on the unperturbed **Identity** dataset to change. This calculation uses the same settings which are subsequently employed for the robustness evaluations under perturbations. To be more precise, this indicates the same parameters n, k, and s that the RP_s@k of single perturbations is measured with. Thus, to determine the performance distribution range, it needs a larger number of baseline runs $N_{total} > n$, with the value of $N_{total} = 20$ in the thesis's experiments.

In order to apply this methodology, the thesis produced baseline distributions for the evaluation parameters used for the *deterministic* perturbations $s = 1$, on all evaluated models, and the *stochastic* perturbations $s = 3$ on *GPT-4o-mini*. These distributions were analyzed in Section 5.2.3, and Section 9.2.3 and revealed significant differences for each model. *GPT-4o-mini* demonstrated comparatively tight, normally distributed fluctuations in Figure 9.4. Other models like *GPT-3.5-turbo* and *Phi-4* showed much greater, multimodal or skewed distributions in the same Figure. This key finding indicated that there cannot be a uniform threshold that specifies "significant change" and that a robustness evaluation always must consider a model-specific threshold to distinguish noise.

While the initial methodology in Section 4.5.1 proposed working with an absolute *Z-Score* of 3.29 to define the significance threshold, the technique could not be applied for the wide-ranging distributions of *GPT-3.5-turbo* and *Phi-4*. For these irregular distributions, the differentiation incorporated the maximum and minimum values of the value distribution. Considering this adaptation, the model-specific baseline was used in Chapters 6 to 9 to provide an orientation for identifying unexpected performance deviations. Only when a perturbation causes performance deviations significantly outside of the distribution and greater than the threshold can we identify a perturbation as likely causing non-robust behavior.

However, it is necessary to note that this approach does not enable a classification of non-robust behavior with complete certainty. It always depends on whether N_{total} accurately reflects the real model fluctuations. Consequently, a larger number of N_{total} runs increases the likelihood that the value distribution is accurate. Other than that, it has been shown that the noise threshold has to be adapted model-specific, because of the varying noise profiles (Figure 9.4). Consequently, this approach might not be fully optimized for a standardized model-agnostic evaluation. Yet, the general strategy remains valid. Performance deviations to perturbed inputs that clearly lie within the baseline distribution can never be confidently attributed to non-robustness, because the model certainly produced the same, or even worse, fluctuations for identical inputs on repeated attempts. Therefore RQ2 can be answered as follows:

Answer to RQ2:
To differentiate between inherent LLM nondeterminism and true robustness deficits, one needs to quantify the individual performance variability of an LLM tasked with identical prompts in repeated runs. By using many runs on unperturbed inputs, under the evaluation conditions relevant for later perturbations, one can build a statistical baseline distribution of expected performance values. With this baseline, one can utilize a statistical strategy suited to the observed characteristics of the distribution to identify outlying performances (*Z-Score*, Maximum, Minimum). When performance under a perturbation deviates greatly from what is expected, it likely signals true robustness deficits.

10.2.3 RQ3: Impact of Feedback Loops on Robustness

Similar to other SOTA code translation systems [Yan+24b; Eni+24], the evaluated system in the thesis leverages the *generate-and-check* pattern for an iterative repair with *feedback loops*. That raised the question for RQ3: "Does incorporating a *feedback loop* strategy impact robustness?"

The implemented *feedback approach* described in Section 4.3.4 utilizes checks for *compilation*, *linting*, and functional equivalence with *fuzzing*. The experiments and analysis in Section 7.2, Figure 7.1, as well as Section 9.4, and Figure 9.9, revealed that *feedback loops* significantly increase the translation performance ($\boldsymbol{RP_s@k}$) on the **Identity** for both *compilation success* and *fuzzing success*. In addition, Figure 7.1 signals that the strongest benefit is due to the first two iterations. Beyond this, the improvements saturate. Moreover, Figure 9.9 shows that the open models *Phi-4* and *Qwen2.5-Coder-14B* only produced comparable performance to *OpenAI's* models *because* of *feedback loops*. In the context of the robustness evaluation, it is worth mentioning that incorporating *feedback loops* also reduced the range of performance fluctuations (Figure 7.1). Using the statistical strategy to distinguish between noise and robustness deficits, this becomes particularly beneficial, as smaller deviations lead to outlying performance.

Besides improving the performance and reducing the variance, the experiments demonstrated that *feedback loops* have a strong impact on robustness. Specifically, they can reduce the magnitude of robustness deficits for perturbations that initially caused significant deviations (Section 7.3, Section 7.4, Section 9.4). Comparing the different RC_s@k values across iterations clearly shows this effect, as the amplitude of values decreases with iterations on *GPT-4o-mini* in Figure 7.7. Similarly, the Figures 9.10 to 9.12 demonstrate the same behavior for *GPT-3.5-turbo*, *Phi-4*, and *Qwen2.5-Coder-14B*.

Analyzing the reasons for each iteration in Figure 7.8 and Figure 9.13 revealed that the *feedback loops* worked best when failures arose due to *compilation* and *linting* issues. The Figures show that the amount of *compilation* and *linting* issues could be reduced with more iterations, yet *fuzzing* issues only reduced slightly in higher iterations. Figure 7.6 confirms this and shows how the perturbations **ConstantInsertion** and **DeadCodeInsertion** initially caused significant performance deviations, but with two to three feedback iterations, improved the performance to the normal range of fluctuations. The investigation in Section 7.4 revealed that both perturbations initially failed because of either *compilation* issues or *linting* issues. By contrast, other perturbations that initially caused non-robust behavior only due to failing *fuzzing* checks (**IdenObfuscator**, **LLMCodeExtraction**), maintained their significant outlying performance also after five iterations (Section 7.4).

Consequently, *feedback loops* can reduce the amount of perturbations that cause true robustness deficits, especially when they are primarily due to translations containing easily fixable issues in *compilation* and *linting*. Comparing Figure 7.4 with Figure 6.4 or Figures 9.6 to 9.8 with Figures 9.10, 9.11, and 9.12 clearly shows that fewer perturbation strategies were causing true robustness deficits after applying five feedback iterations. However, with *feedback loops* improving the performance and reducing the value range of the baseline distribution, it could also be observed that certain perturbations did not improve comparably to the baseline (**CamelCase** for *GPT-3.5-turbo* in Section 9.4.1). This led to the perturbation causing non-robust behavior, as they are not in the expected range for the system with *feedback loops*. Therefore, *feedback loops* do not always improve robustness and reliability for all scenarios. However, since they consistently improve the general performance and reduce the magnitude of most deviations for all evaluated models, they present a highly beneficial technique for the robustness and performance of an LLM-based code translation system. Based on the gathered information RQ3 can be answered as follows:

Answer to RQ3:
Yes, the *feedback loop* strategy positively impacts the robustness of LLM-based code translation systems. They reduce the magnitude of performance deviations caused by perturbations and significantly improve the general performance while simultaneously decreasing the range of the model's fluctuations. They work best when issues are due to translations failing *compilation* or *linting* checks and have limited capabilities when issues are stemming from *fuzzing* differences. So while they usually reduce the gap between baseline and perturbed inputs, they do not necessarily erase all robustness deficits for fundamentally difficult perturbations.

10.2.4 RQ4: Correlating Semantic Similarity and Robustness

With RQ4, the thesis motivated an investigation into whether identifiable robustness deficits are primarily driven by the magnitude of change a perturbation causes, or whether the effects are specific to the strategy itself. Non-robust behavior for perturbations highly similar to the **Identity** is more concerning than for perturbations that produce completely different inputs. This led to RQ4 asking: "What is the correlation between semantic similarity and perturbation-based robustness?" If there is a clear correlation, the semantic similarity measure can be leveraged to predict the likelihood of robust performance and additionally be used to reveal perturbation strategies that cause deficits with only a few but pivotal changes.

To approach the question, the thesis employed embedding-based *cosine similarity*, quantifying similarity scores to the unperturbed **Identity**. Small examples showed that this approach yielded more desirable similarity scores for this context than other lexical or matching approaches (Section 4.4.4). The intuition was that an embedding-based measurement is better suited for quantifying semantical changes and accurately reflects similarity for perturbations, adding or changing comments, modifying variable names, or formatting the code style. Specifically, *OpenAI's ada-002* model [Ope22c] that has also been trained for vectorizing code inputs seemed a good choice for this challenge. While *cosine similarity* can in theory produce values between -1 and 1, in reality, the value range is much smaller, as shown in the examples in Section 4.4.4. Thus, to enable a better interpretation and distinguish between heavily modified and sparsely modified inputs, the thesis established a baseline distribution by computing pairwise similarity scores of all files of the dataset. Subsequently, this value distribution shows similarity scores for files that are not semantically similar, so values that are higher than this distribution are likely to show similar scores. Again, the thesis used the *Z-Score* to classify whether perturbations are "relevant" since they produced highly similar input variations, or whether they are less relevant as the changes were too drastic.

Applying this technique signaled that most perturbations that target code and comments yielded *cosine similarity* scores, which suggest high semantic similarity. Only the **ABC**, **ConstantInsertion**, and **DeadCodeInsertion** perturbations resulted in seemingly low similarity scores. Furthermore, the similarity scores on the instructions were harder to interpret, as their scores were more strongly impacted by perturbations. By summarizing these findings, the thesis concluded that similarity scores with this approach are heavily dependent on the number of tokens of the compared input. For the context of this thesis, this is not optimal, as smaller files might show stronger similarity deviations than larger files, while still having undergone the same semantic modification. This makes a correlation analysis less meaningful, as it does not fully reflect the desired concept of semantic similarity. Moreover, with **ABC** producing the smallest similarity scores for *deterministic* perturbations on the code part, embedding-based similarity suggests being sensitive to modifications of pivotal tokens. This perturbation only replaces identifier names with a single character, not changing the control flow or structure of the code. While this finding was unexpected, it shows that this approach may capture differences between a human conception and an LLM's conception of the inputs.

To gather the information necessary to answer the RQ Sections 8.3 to 8.4 and Section 9.5, visualize the correlation between $\boldsymbol{RP_s@k}$ and *cosine similarity* in *scatter plots*. In case of a clear correlation, these scatter plots would have shown a linear progression. Specifically, with increasing similarity, the deviation from the baseline would have gotten closer. However, while this is slightly noticeable for the translation statistics without *feedback loops*, the correlation vanished completely for all evaluated models after feedback iterations were applied. For a less subjective measurement, the thesis additionally incorporated the *Pearson correlation coefficient* [SBS18]. Nonetheless, this only confirmed the results that could be seen visually. For instance, the already mentioned **ABC** perturbation resulted in the least similarity for perturbations on code, yet the translation performance of all models was mostly inside the expected value range (Figure 8.2, Figure 9.14, Figure 9.15 or Figure 9.16). Similarly, **DeMorgan** is suggested as a highly similar perturbation, yet it caused significant outlying performance for *Qwen2.5-Coder-14B* in Figure 9.16. Consequently, with the proposed methodology, there is no clear correlation between robustness and semantic similarity that could be used for an a priori prediction. Whether this is because of the used *cosine similarity* not being the best semantic similarity measure, or rather because a model's robust performance is highly individual and not predictable, remains an open question for future work, which will be discussed in Section 10.4. With the knowledge of the thesis, the RQ can be answered as follows:

Answer to RQ4:
Using embedding-based *cosine similarity* suggested that there is no strong linear correlation between similarity and translation robustness, especially when *feedback loops* are active. While certain scenarios and subsets show minor trends, *cosine similarity* was not sufficient to reliably predict which perturbation would cause robustness deficits.

10.2.5 RQ5: Robustness Across Models

While the conclusions of the other RQs already contained information for all models, the thesis initially examined robustness for *GPT-4o-mini* in detail (Chapters 6 to 8). In Chapter 9 it extended the evaluation to other models to gather the information for RQ5: “Are robustness results consistent using different LLMs?” By extending the analysis, the thesis investigated whether code translation robustness findings for *deterministic* perturbations are transferable to other models or if they are model-specific.

Chapter 9 first demonstrated that the baseline performance varies strongly across models. No model turned out to be superior in all scenarios, and determining the best model depended on the number of translation attempts (Figure 9.1). Moreover, Figure 9.2 illustrated that some models can translate files that other models fail at and vice versa. Lastly, it could be observed that the noise profile of the models does vary greatly. *GPT-4o-mini* shows a tight, normally distributed profile, whereas *Phi-4*, or *GPT-3.5-turbo*, show skewed or multimodal value distributions, subsequently influencing the trustworthiness of the *Z-Score*.

Focusing on the robustness findings of the different models reveals that they follow some general trends, but are mostly highly individual to the model. In detail, after applying the *feedback loops*, it could be observed that **IdenObfuscator** consistently was a perturbation where the models showed comparably worse translation performance than for the *Sampled Identity* baseline (Figures 6.1a, 9.6, 9.7, 9.8). Additionally, it could be observed that the models predominantly performed robustly under the *deterministic* perturbations. This highlights that there are some prominent consistencies for all models. However, the cross-model comparison also revealed different significant perturbations for different models beyond **IdenObfuscator**. To be more precise, *GPT-4o-mini* exhibited non-robust performance improvements for **Translation-GER-Comments** and **CodeFormat-Mozilla** (Figure 5.5b), whereas *GPT-3.5-turbo* signaled non-robust improvements for **CamelCase** (Figure 9.6), and *Phi-4* for all perturbations targeting comments (Figure 9.7). Conversely, *Qwen2.5-Coder-14B* showed significant performance deviations for **DeMorgan** (Figure 9.8). So while a predominant robust performance for most perturbations could be observed, the examination highlights that the actual robustness for certain strategies has to be evaluated model-specific.

Besides the general robustness, it could be observed that the conclusions for *feedback loops* are consistent for all models. Namely, they reduced the magnitude of non-robust deviations without fully ensuring robust performance for all perturbations after all iterations (Section 9.4). This underlines that this technique is beneficial for all LLMs in code translation. Furthermore, the discovery that the *feedback loops* work best when repairing *compilation* and *linting* issues turned out to be consistent (Figure 9.13). Another surprising finding was that the open models *Phi-4* and *Qwen2.5-Coder-14B* benefited more from the technique than the models from *OpenAI*, since their initial performance was significantly worse.

Similarly, correlating *cosine similarity* and translation success for all models confirmed the prior findings that there is no reliable correlation between semantic similarity and robustness. It remains a question whether this is due to the *cosine similarity* approach not accurately reflecting the actual similarity of the input variations, or the models simply not being predictable just with the knowledge of input similarity (Figures 9.14 to 9.16).

Concluding the gathered information to select the “optimal” model for the code translation system in Figure 9.17a and 9.17b demonstrates that the decision is multi-faceted. The figures show that the optimal choice of the model depends on the number of translation attempts, the acceptable fluctuation range, the robustness against perturbations, and the performance. There was no model that was superior in all these facets, leading to the decision being strongly dependent on priorities and requirements.

Summarizing this information RQ5 can be answered as follows:

Answer to RQ5:
Robustness results are not fully consistent across different LLMs. Specifically, the robustness of single perturbations, as well as the models' noise profile, can vary greatly and have to be investigated for each model in particular. Nonetheless, the models predominantly performed robustly under perturbations, while having a weakness for **IdenObfuscator** in common. Moreover, the impacts of *feedback loops*, as well as the absence of a correlation between *cosine similarity* and robustness, remained consistent. However, the differences in performance and sensitivity to perturbation strategies ultimately lead to performance-robustness trade-offs that are highly model-specific. Therefore, robustness has to be evaluated individually for each LLM considered for a code translation.

10.3 Threats to Validity

This section discusses potential limitations of the proposed framework and the discoveries and conclusions that could be found by applying it. These limitations can influence the validity of the conclusions about the robustness of LLM-based code translation systems and the RQs. The threats to validity are subsequently distinguished between *internal* validity and *external* validity. Furthermore, the section provides a critical reflection on specific framework elements that, in light of insights gained from the evaluation, no longer appear optimal.

10.3.1 Internal Validity

Internal Validity refers to whether the discoveries and conclusions are truly because of the examined reasons, or if and to what extent they fall victim to limitations of the methodology.

Differential Fuzzing as Approximation of Functional Equivalence As mentioned before, the utilization of *differential fuzzing* to verify *functional equivalence* is only an approximation of *functional equivalence*. It cannot guarantee equivalence, conversely to other approaches like a formal verification, which is not fully autonomous and was used by *VERT* (Section 2.4.2). Moreover, *differential fuzzing* depends on the configured *fuzzing time*, and one could argue that the chosen fuzzing time of 15 seconds might not be enough. However, different fuzzing times have been manually tested beforehand, highlighting that *counterexamples* are identified rather quickly, and longer fuzzing times did not reveal any other *counterexamples*. Nonetheless, it should be noted that the approach comes with the risk that subtle functional dissimilarities remain undetected, potentially leading to an overestimation of the system's capabilities.

Implementation Errors in the Framework Both the *differential fuzzing* as well as the automatic generation of perturbations are not trivial and might involve errors or unexpected behaviors in certain edge cases. With respect to fuzzing, the framework documented *fuzzing exceptions* and *fuzzing setup* errors to incorporate them into the robustness interpretation in Chapter 6. This led to excluding the **Translation-KOR-Code** from the evaluation, as the fuzzer was not able to process identifiers with Korean characters. All perturbations showed some amount of errors in Appendix A.4.1, meaning that they could potentially have influenced the conclusions about robustness. However, the overall percentage of errors among all translation attempts was small, and it was shown that these errors have never been the major reason for problematic or non-robust translations. So the actual risk of these errors impacting the conclusions is rather small, especially when evaluating with $n = k = 5$, which will also be discussed in this section.

Besides the validity of fuzzing, another important threat is the validity of perturbations. While all perturbations incorporated a *syntax check*, to directly detect syntax errors that prevent the application of the fuzzer, the framework did not check for *semantic equivalence* of perturbations. While it would have been possible to use another *differential fuzzer* that verifies *functional equivalence* and therefore *semantic equivalence* between **Identity** and the perturbed Code, the decision was made not to include it, as it would have exceeded the scope of this thesis. The implemented perturbations were chosen because they should not change the semantic equivalence by definition. Nonetheless, without a tailored check, the perturbation cannot be guaranteed to adhere to the semantical equivalence requirement (Section 4.2.1).

Another threat that is noteworthy in this context is that **IdenObfuscator** produced above-average percentage *fuzzing setup* errors. Upon close investigation, these errors are due to the perturbation creating the function name "_" that is valid in *C*, but creates a syntax error when translated into *Rust*. This makes a successful fuzzing impossible when such a function name is created by the perturbation. However, in Appendix A.4.1 it has been shown that none of the non-robustly translated files were affected by these issues for *GPT-4o-mini*, meaning that the LLM failed to create a functionally equivalent translation and was not limited by the capabilities of the fuzzing at this point. Therefore **IdenObfuscator** indeed preserves to be a perturbation causing true non-robustness. A similar problem was observed for **LLMVariableImprove**, which produced identical function names with different casing strategies. This was not an issue in *C* but created syntax errors in *Rust*. Section 6.2.3 points out that this issue did not significantly impact the robustness interpretation of this perturbation, yet it should be noted in this context.

Likewise, Appendix A.4.2 displayed that the code translation system had higher error rates for all perturbations upon using the other LLMs. This could have resulted in an underestimation of a model's overall translation performance. However, Section 9.6 concluded that **Identity** also showed an elevated error rate, which therefore means that the findings about robustness remain relevant. Furthermore, since repeated attempts seemed to solve these errors, the interpretation was that this increased error rate is primarily due to a weakness of the models and not a weakness of the framework itself.

By incorporating the documented errors into the interpretation of the results, we can minimize the risk of misinterpreting results due to framework errors instead of genuinely identified robustness issues.

Selection of Metrics and Parameters Another factor that influences the interpretation of the assessment of the used parameters for RP$_s$@k. The thesis mostly focused on an evaluation with $n = k = 5$ for each perturbation. When evaluating with $n = k$, the metric results in full correctness ($RP_s@k = 1$) as soon as one of the n attempts succeeds. Therefore, it masks the true probability of generating a successful translation for only one attempt. However, the decision was to use these parameters, as they are less impacted by errors of the translation system. Furthermore, as the results of the thesis have shown, for the best performance, it is necessary to work with repeated translation attempts. Choosing $n = k = 5$ reflects this application with five attempts. Nonetheless, including $k = 1$ would have also been beneficial in hindsight, but would have resulted in an even more elaborate discussion. Besides this, evaluating stochastic models always raises the question of whether there are enough data points n to enable a meaningful interpretation. Figure 9.1 details, that not all models stagnate completely within $n = 20$ attempts. So, increasing the number of baseline and perturbation runs could lead to a more statistically safe estimation of robustness. However, as each additional run per perturbation significantly increases the total number of LLM calls, it also significantly impacts the overall cost, regarding runtime and API-usage. Considering this, Figure 9.1 shows that within $n = 5$ the models begin to saturate, underlining that five is a good trade-off between cost and performance.

Statistical Certainty of Single Perturbations The framework judges robustness by comparing the results of a perturbation against the performance of the unperturbed baseline. Specifically, it uses a single aggregated $\mathrm{RP}_s@\mathrm{k}$ value under a perturbation and compares its position relative to the unperturbed baseline. This comparison only involves a single data point and therefore cannot guarantee that this specific data point is representative of the perturbation's own underlying performance distribution. An ideal, statistically sound test must also incorporate the fluctuations under perturbations to prevent perturbations from being incorrectly classified as robust, consequently comparing statistical significance between two distributions. However, given the large amount of perturbations, files, and feedback loops, this would have resulted in a tremendous increase of LLM-calls, which was infeasible due to cost and time restrictions. Nonetheless, the way the results were interpreted is still relevant. Since the decision was to use the stricter *Z-Score* of $|3.29|$ as a threshold, outlying data points of a perturbation remain highly likely to be genuinely caused by the variation itself.

Causality Feedback Loops The thesis concluded that *feedback loops* improve robustness of LLM-based code translation systems. The visualizations in Chapter 7 and Section 9.4 showed that the iterative repairs improved the performance and decreased the range of deviations found for perturbations. However, it cannot be guaranteed that these findings are caused by the feedback itself, or rather by the fact that the LLM had another attempt to translate the code correctly. It is hard to isolate the causal effect of the prompted feedback from simple repetition.

10.3.2 External Threats

External threats include factors that could influence the generalization and transferability of the thesis's conclusions. Therefore, these threats describe potential issues that could prevent the conclusions from being relevant to other works in the same area, or applications of LLM-based code translation systems.

Limitations of the Dataset The evaluation was conducted on only 50 files. While it was aimed to increase the diversity by sampling from different sources (i.e., proprietary automotive code, open source libraries, and competitive programming tasks), it remains the question whether this small exemplary dataset covers the broad diversity of legacy *C* codebases that could be translated into *Rust*. Similarly, the files predominantly originated from *embedded automotive code*. It is unclear whether the findings are directly transferable to other application domains of *C* and *Rust*.

Limitations of the Models Another factor that influences the transferability of the results is the chosen models. As RQ5 points out, results are not directly transferable to other models. The evaluation only covers the models *GPT-4o-mini*, *GPT-3.5-turbo*, *Phi-4*, and *Qwen2.5-Coder-14B*. Therefore, the evaluation misses a robustness assessment of *reasoning models*, which, according to SOTA benchmark leaderboards, produce the best results in logical tasks like code generation [Cod25b]. During the development of the framework, access to affordable reasoning models was limited. Consequently, due to the cost and runtime, the decision was not to include reasoning models.

Furthermore, the evaluation has only been conducted with temperature-based sampling configured with a temperature of 0.7. The results might look different when using another sampling strategy or other temperature values.

Lastly, because of the limited time-scope, the evaluation for *Qwen2.5-Coder-14B* was only conducted with $n = 1$ per perturbation, clearly impacting the trustworthiness of the interpretations for this model.

Limitations of the Use Case The thesis focused on evaluating *C* to *Rust* translation. Therefore, the results may not generalize to other LLM-based tasks. They might not even generalize to translation tasks between other language pairs. However, evaluating this was not the focus of this thesis.

10.3.3 Critique on Framework Elements

With the gathered knowledge from the evaluation, there are other aspects of the framework that go beyond internal and external validity.

The first aspect is the *Z-Score* based robustness threshold. While this approach worked well for the normally distributed baseline of *GPT-4o-mini* and *Qwen2.5-Coder-14B*, it was not applicable to *GPT-3.5-turbo* or *Phi-4*. The necessity to adapt the threshold, incorporating minimum and maximum values of the distribution, negatively influences the comparability and standardization of the robustness classification between models. The reliability of this methodology strongly depends on the noise profile of the evaluated model.

Besides the weakness of the *Z-Score* RQ4 concluded that there is no correlation between semantic similarity and robustness. However, this conclusion is limited to the perturbations implemented in the framework. Since these perturbations were initially designed not to produce highly different or semantically unequal versions, the conclusion might be different with another set of perturbations. Furthermore, the evaluation only presented the results for one embedding model *ada-002*. Other models might yield other cosine similarities. At this point, it is not definitely clear whether robustness is truly unpredictable through similarity of inputs, or whether the chosen model and perturbation subset led to this conclusion. However, it is worth mentioning that multiple different embedding models were tested during the development of the framework. Doing this, it could not be observed that an overly drastic similarity difference existed when comparing the *scatter plots*. The decision for *ada-002* was due to it being an embedding model of *OpenAI* that might be better suited to an evaluation primarily conducted on *GPT-4o-mini*.

10.4 Future Work

The critical reflection on the threats to validity of this thesis not only highlights the limitations of this work but also reveals potential areas for improvement for future works. These areas can be distinguished into three groups: *(i)* refinement of the evaluation framework, *(ii)* extending the scope of the evaluation, and *(iii)* incorporation of advanced mechanisms.

10.4.1 Refinement of the Evaluation Framework

The previously discussed critique of framework components in Section 10.3.3, implies the improvement of the proposed evaluation framework itself.

Update the *Z-Score* Approach Using *Z-Score* to distinguish between noise and true non-robustness is not trustworthy for non-normally distributed noise profiles. Furthermore, the thesis identified the limitation of assessing the robustness by a single data point under a perturbation. Future works should investigate alternative approaches to distinguish between noise and non-robustness.

They could update the proposed methodology and use a statistical approach that is statistically robust, also for non-normally distributed noise profiles and allows comparing value distributions under a perturbation, like the non-parametric *Wilcoxon-Mann-Whitney-Test* [Wil45] for example. This would increase the certainty of the results. In addition, by removing the need to select a suitable robustness threshold for a given noise profile (*Z-Score*, *Maximum*, or *Minimum*), it would become a more standardized interpretation of the translation statistics.

Measuring Semantic Similarity The employed *cosine similarity* with *ada-002* did not reliably correlate with robustness as discussed in RQ4. Future works could investigate whether this is due to the limitations of *cosine similarity* reflecting the *semantic similarity*, or rather an unpredictability of LLMs. A possible starting point could be to utilize approaches that are specifically designed for code comparisons, like Zhou et al.'s *Code-BertScore* [Zho+23]. Their approach also relies on embeddings, but they do not solely convert an entire file into an embedding vector. Instead, they employ a more fine-grained approach using pairwise cosine similarity between "non-punctuation code tokens".

Improving Step I of the Framework The prior section detailed that the framework missed on directly verifying *semantical equivalence* between **Identity** and the perturbed file. To ensure this requirement, future works could extend the framework to automatically incorporate *differential fuzzing* on the **Identity** and the perturbed version. Preventing the generation of perturbations that do not adhere to the semantic equivalence requirement and therefore would not reflect robustness, but rather the framework's abilities to translate an entirely different task. Moreover, the framework should be extended to include a mechanism to prevent the generation of function names that are invalid in *Rust*, which blocks the framework from applying *differential fuzzing*. A simple blacklist including "_" could have reduced **IdenObfuscator** from producing *fuzzing setup* errors. Similarly, a check that prevents producing equal function names with different casing strategies would have been beneficial for **LLMVariableImprove**. Furthermore, since the thesis concluded that robustness among perturbations is not transferable to other perturbation strategies with similar characteristics, extending the number of perturbations would be a way to cover a broader range of input variations. Specifically, such perturbation strategies could focus on the already known difficulties when translating *C* to *Rust* (i.e., introducing global variables, unions, goto statements, and more, see Section 2.1.2).

10.4.2 Extending the Evaluation Scope

To improve the generalization and transferability of the results, future work could increase or change the scope of the evaluation.

Other Benchmark Datasets Future works could evaluate robustness on larger datasets, with more files that cover a broad range of real-world relevant *C* code. Since the results under **LLMCodeExtraction** signal a weakness for code snippets with nested functions and complex control flows, the dataset should also include such algorithmically complex examples to investigate if this is an inherent weakness, or whether it was caused by the **LLMCodeExtraction** adding unnatural code variations.

Evaluate other LLMs The thesis did not investigate the robustness of reasoning models. During the design of the thesis's experiments, such models were pricey and not affordable for the tremendous amounts of LLM calls in a robustness evaluation. However, recent innovations like *GPT-o4-mini* [Ope25b] show a trend that reasoning models are becoming more affordable, making them another option for a robustness evaluation. It would be interesting to see whether reasoning models with their improved capabilities in logical thinking are even more robust than normal LLMs.

Robustness in Translating Other Language Pairs The thesis entirely focused on evaluating the robustness of an LLM-based *C*-to-*Rust* code translation system. Another valuable contribution would be to investigate whether the discoveries of this work are transferable to the translation of other relevant language pairs like *COBOL-Java* [Gan+24], *Java-Kotlin* [Tao+24], and others. The general methodology is directly applicable to other languages. However, one would need a *differential fuzzer* tailored to the desired language pairs. Additionally, languages that are not supported by *tree-sitter* [Bru+25] would need more implementation, as the perturbation process heavily relies on this package.

Optimizing Sampling Strategies The thesis only employed temperature-based sampling with a fixed temperature of 0.7. A valuable contribution would be to investigate the impact of different sampling strategies on the robustness. Such an investigation may reveal the optimal sampling configuration to optimize the trade-off between robustness and performance.

10.4.3 Advanced Translation-Mechanisms

The findings of this thesis sparked creativity and justify trying other translation techniques, which may improve robustness and translation performance.

Alternative Feedback Loops Strategies To empirically prove that the effects of RQ2 were due to the feedback and not merely due to repeated translation, future work should incorporate a direct comparison that includes a similar amount of iterations in case of failure but without giving the model feedback about the failing reason. Moreover, the implemented *feedback approach* could be extended to include more context and better feedback for fuzzing-related failures. This could account for examining whether feedback loops limitations in fuzzing failures are due to the feedback itself or the incompetence of the LLM fixing logical errors.

Preprocessing the Data The thesis observed that the models *GPT-4o-mini* and *Phi-4* are sensitive to perturbations on comments in German and produce significantly improved translation performance. To solidify this finding, it may be interesting to preprocess the dataset to apply such a perturbation to replace the **Identity** to only include German comments. According to the results of the thesis, this should improve performance, and might be more robust than the current **Identity**? Additionally, we saw that perturbations like **IdenObfuscator** and **LLMVariableImprove** can produce perturbations that are valid for *C* but cannot be translated into *Rust* because of invalid function names. It would be good to preprocess the dataset or files to be translated beforehand, to update problematic function names in *Rust* to ensure that such an issue is not the reason for a non-fuzzable translation.

Ensemble Translations Figure 9.2 suggests that different models have different strengths when translating files. One model might successfully translate a file that another does not, and vice versa. Considering this, a cheap way of improving the general performance of a code translation system could potentially be to employ ensemble approaches that combine different models into the translation pipeline. One way would be to iteratively try another model in case of failure, starting with the cheapest model. In case none of the ensemble models translate the file successfully, one could start with feedback iterations, starting with an additional iteration for the cheapest model, and trying the more expensive models, if it does not work. Another way that would be inspired by Figure 9.1 might be to start with a model that has high one-shot performance without *feedback loops*, *GPT-4o-mini* for example, and in case of failure, using a model that showed higher sensitivity through *feedback*, like *Phi-4*. It would be interesting to see whether such techniques could lead to improved performance. Lastly, this would also pose the question whether such an approach is more or less robust than translations with only one model.

Incorporating Safe Rust into the Measurement The thesis only evaluated robustness according to *compilation success* and *fuzzing success*. While *fuzzing success* is the ultimate goal of a successful code translation, to additionally quantify the received benefits through an LLM-based translation compared to rule-based approaches, it would be necessary to quantify the success ratio of translating into *safe Rust*. Consequently, this additional metric would raise the robustness question, whether perturbations would hinder LLM-based systems from translating into *safe Rust*. However, such a metric should also account for feasibility, since not all concepts of *C* code are translatable to *safe Rust* [Theb].

10.5 Are LLM Translators a Benefactor or an Enemy?

The thesis started with Miguel Sáenz's statement that a translator can either be a *"benefactor of humanity"* or a *"veritable public enemy"* [MW21]. Considering today's rapid progress and acceptance of LLM-based SE, this raises the question of whether modern LLM-based translation systems are Sáenz's *"benefactor"* or rather perform like a *"veritable public enemy"*. Considering *DARPA's* call for *C-to-Rust* translation solutions [DAR24], Sáenz's quote becomes especially relevant for this particular translation task. The fact that *C* is used across many legacy and safety-relevant systems, even though it is susceptible to *memory-safety*, makes a translation into a *memory-safe* language like *Rust* highly desirable. However, traditional automatic translation approaches fail to leverage *Rust's* idiomatic safety features, whereas manual conversions are costly and prone to result in functionally inequivalent translations [Li+25], potentially introducing new bugs to an originally working system. LLMs and their "emergent abilities" in processing natural language and code present a promising alternative to these approaches. The thesis motivation highlighted that even though there is an enormous potential for an LLM-based modernization of old codebases [Yan+24b; Yan+24a], the area shows relevant research gaps. Specifically, the literature lacked a comprehensive robustness evaluation of such translation systems, which assesses their robustness to the variations and uncertainties of real-world codebases. Prior works that evaluated the robustness of LLM-based code generation only partially addressed this and focused on other use cases, not real-world relevant benchmarks, or purely on natural language perturbations [Wan+23; Yan+23a; Mas+23; Imp+25].

This thesis aimed to close this gap. Its main contribution is the development and application of a comprehensive robustness evaluation framework tailored to LLM-based *C* to *Rust* translation. The proposed framework includes components for *(i)* diverse and code-focused perturbations that cover variations of varying complexity levels *(ii)* an preexisting SOTA code translation system incorporating a *generate-and-check* pattern that automatically verifies *functional equivalence* through *differential fuzzing* and applies auto-repairing *feedback loops*, as well as *(iii)* a multi-dimensional evaluation concept, leveraging existing and novel metrics (RP_s@k, RC_s@k) with statistical baseline analyses, that aims to be a reference when distinguishing between inherent model noise and true robustness deficits (RQ1, RQ2).

Applying this framework on modern LLMs like *GPT-4o-mini*, *GPT-3.5-turbo*, *Phi-4*, and *Qwen2.5-Coder-14B* gave a detailed view on their robustness:

Against prior beliefs and older code generation robustness works modern LLMs surprisingly signal predominantly robust behavior under most real-world relevant input variations, reflecting changes in formatting, identifier or comment-based refactorings, as well as control flow variations (Chapter 6, Chapter 9). The evaluation clearly showed that LLM-based translation has the potential to act as a *"benefactor"* when facilitating the tedious task of manual translations.

However, the code translation system was not entirely robust for all scenarios. The models revealed weaknesses in processing certain perturbations (**IdenObfuscator**, **LLM-CodeExtraction**) and additionally exhibited model-specific sensitivities (RQ5). Therefore, without a comprehensive evaluation, LLM-based code translation can indeed act unexpectedly and unpredictably, consequently behaving like a potential *"enemy"* that does its job not reliably.

A pivotal feature that pushes LLM-based code translation systems towards behaving like a *"benefactor"* is *feedback loops*. The experiments showed that they not only significantly increase the overall translation performance but also improve the robustness. They proved especially helpful when repairing compilation and linting errors, which ultimately led to a reduction of the magnitude of non-robust performance deviations (RQ3). Nonetheless, the results suggest that their abilities are limited when problems arise due to deeper logical errors that lead to fuzzing counterexamples.

In addition, the thesis shows the individuality of the models. We saw that among the evaluated models, no model was "the" most robust. Instead, defining and selecting the best model is a trade-off between performance, robustness, and inherent noise profile, while accounting for how many translation attempts one can afford in terms of price and compute time (Figure 9.17). Likewise, the thesis came to the conclusion that it is not possible to predict robustness by quantifying semantic similarity with *cosine similarity* (RQ4).

Consequently, all findings move towards the necessity of model-specific and task-specific evaluations, with specifically tailored perturbations to prepare and evaluate a translation system for practical usability. That clearly aligns with the concerns of Nezhurina et al. [Nez+24], stating that LLMs abilities should not be overestimated when generalizing prior benchmark results to seemingly similar tasks.

So are LLM-based translator *"benefactors"* or *"enemies"*? Based on the thesis discoveries, the answer is that they are both, or at least have the potential to be both. They clearly are not marvelous systems that reliably cope with all input variations, leading to infallibly robust translations. Yet, they are also not true *"veritable public enemies"*, that always act unexpectedly when prompted with potentially relevant input variations of practical environments. Rather, they are predominantly robust systems with only minor weaknesses. Therefore, their dangers and benefits are closely related to how we use and evaluate them. With a comprehensive robustness evaluation like this, we can reveal the particular weaknesses that may be relevant for a specific use case and reduce the system's potential of unexpectedly becoming an *"enemy"*. In addition, with strategies like *feedback loops*, we can support it even more so to act as *"benefactor"*. The thesis showed that robustness is not an optional feature, but rather another dimension beyond simple translation performance. Only by continuously assessing the robustness of LLM-based translation systems can we ensure that these systems become reliable tools in the complex yet desirable process of software modernization.

A Appendix

A.1 Perturbation Strategies

Table A.1 demonstrates perturbations applied to an exemplary prompt, where the original prompt is defined as follows.

Exemplary Identity Prompt

Instruction: Translate the following C code to Rust. Keep all identifiers exactly as they are.

Code:

```c
#include <stdio.h>

// Recursive function to calculate Fibonacci numbers
int calc_fibonacci(int n) {
    if (n <= 1) { return n; }
    return calc_fibonacci(n - 1) + calc_fibonacci(n - 2);
}

int main() {
    int n = 10, i;
    for (i = 0; i < n; i++) {
    printf("%d ", calc_fibonacci(i)); }
    return 0;
}
```

Moreover, this section details the functionality of the implemented perturbation strategies and shows that they adhere to the requirements of a perturbation.

A.1.1 Implementation Details

Since programming languages follow strict syntactic rules, implementing perturbation strategies on code or comments is not a trivial task, which can be demonstrated by a small example. Although replacing variable names seems to be an easy task, without the use of tools, it can get complicated and error-prone. Replacing a variable does not mean to replace a single string, it means to replace all occurrences of this string, where the string belongs to the scope of the variable.

Think of an iteration variable i in a *for-loop*, for example. Replacing all occurrences of i in the code part of the prompt would not be accurate, since it could include multiple definitions of i, outside this specific loop. Instead, one would have to replace all occurrences of i, inside the scope of the *for-loop*.

This small example shows what goes into the supposedly easy task of replacing a variable name. Fortunately, some tools simplify such operations. To be more precise, the thesis utilized *tree-sitter* [Bru+25], a Python package that enables parsing code into an

Perturbation	Target	Parameter	Result
Backtranslation	Instruction	*EngGerEng*	identifiers → the identifiers
Butterfinger	Instruction	Prob: 0.05	Translate → Transmate
ChangeCharCase	Instruction	Prob: 0.3	Translate → tranSLaTE
Concretizer	Instruction	Min: 3 Max: 5	+ The code contains a for loop . . .
Translation	Instruction	German	Translate → Übersetzen Sie
Translation	Instruction	Korean	Translate → 번역하십시오
Backtranslation	Comments	*EngGerEng*	to calculate → for calculating
Butterfinger	Comments	Prob: 0.05	Fibonacci → Fibonscci
ChangeCharCase	Comments	Prob: 0.3	Recursive → REcursiVe
LLMCommentInsertion	Comments		+ // . . . essential to terminate recursion . . .
RemoveComments	Comments		− // Recursive function . . .
Translation	Comments	German	Recursive function → Rekursive Funktion
Translation	Comments	German	Recursive function → 재귀 함수
CodeFormat	Code	Style: Mozilla	`if (n <= 1){ return n;}` → `if (n <= 1)`
ABC	Code		`int calc_fibonacci(int n)` → `int b(int a)`
Backtranslation	Code	*EngGerEng*	`int calc_fibonacci(int n)` → `int calculate_fibonacci(int n)`
Butterfinger	Code	Prob: 0.05	`int calc_fibonacci(int n)` → `int calc_fibonaxci(int n)`
CamelCase	Code		`int calc_fibonacci(int n)` → `int calcFibonacci(int n)`
ChangeCharCase	Code	Prob: 0.3	`int calc_fibonacci(int n)` → `int cAlc_fibonacCi(int n)`
IdenObfuscator	Code		`calc_fibonacci(int n)` → `calc_(int n)`
LLMVariableImprove	Code		`int calc_fibonacci(int n)` → `int get_fibonacci(int n)`
PascalCase	Code		`int calc_fibonacci(int n)` → `int CalcFibonacci(int n)`
SnakeCase	Code		- (already is snake_case)
Translation	Code	German	`int calc_fibonacci(int n)` → `int berrechne_Fibonacci(int n)`
Translation	Code	Korean	`int calc_fibonacci(int n)` → `int 피보나치_계산(int n)`
ConstantInsertion	Code		+ `const int DEFAULT_TIMEOUT = 30;`
DeadCodeInsertion	Code		+ `static inline int multiply(int a, int b) {return a*b}`
IncludeCommentAdder	Comments	*used_calls*: False	<stdio.h> → <stdio.h> /*getchar()*/
LLMCodeExtraction	Code		`for (i = 0; i < n; i++)...` → `print_fibonacci_sequence(n);`
FunctionSignatureChange	Code		`int calc_fibonacci(int n)` → `int calc_fibonacci(int n, int a)`
ForWhileSwitch	Code		`for (i = 0; i < n; i++)...` → `while (i < n)...`
ConditionSwap	Code		`if (n <= 1){ return n;}` → `if (1 >= n){ return n;}`
ConditionDup	Code		`if (n <= 1){ return n;}` → `if (n <= 1 && true){ return n;}`
DeMorgan	Code		`if (n <= 1){ return n;}` → `if !(n > 1){ return n;}`

Table A.1: Perturbations applied to Exemplary Identity Prompt. The result column shows one of the effects a perturbation had on the example prompt.

AST-representation. The AST representation enables easier modification of certain code structures, syntax is represented in nodes, which subsequently can be modified.

The *tree-sitter* package comes with a query language, which enables querying for specific tree-nodes. Going back to the variable name replacement, one can easily query for the identifier node that represents the variable and replace its name. This ensures that only relevant occurrences of the variable are changed, rather than when implementing it with a naive string replacement.

Despite using the abstraction layer of *tree-sitter*, perturbing code syntactically correct remains a nontrivial task, and unforeseen edge-cases can still lead to producing errors.

Hence, the framework involves performing a syntax check after a perturbation has been created, by using the diagnostic tools of *clang*-compiler [LLVa]. Incorporating such syntax check is not only a convenient feature during the implementation process, instead it is also a valid way to ensure that a perturbed dataset adheres to the requirement of being syntactically correct.

In addition, the syntax check also helps when perturbing the code using LLMs. The nondeterministic nature of LLMs can lead to producing incorrect results, although the model always produced the expected results during development. Similar to the *generate and check* pattern of the code translation system, the perturbation process utilizes a feedback loop strategy for LLM-based perturbations. As long as the model's perturbation response fails the syntax check, the model is re-prompted, including the generated code with the *clang* diagnostic error description. While this does not guarantee a syntactically correct result, it increases the chance of getting a correct result. However, because of no correctness guarantee, limiting the number of retries is necessary for such a process. In detail, the framework uses a maximum of five retries by default, as this showed promising results during the development process of the framework.

A.1.2 Perturbations Functionality in Detail

Identity **Identity** is the baseline "perturbation" that performs no modifications to the instruction nor to the code part of the prompt. The commonly used term *Sampled Identity* refers to the mean value of the computed baseline distribution, which is explained in Section 4.5.1.

Backtranslation **Backtranslation** translates text into a target language and then back into the original language, producing a paraphrased version of the original. Paraphrasing is highly relevant as it reflects that different persons can produce different texts or different code for the same task. This technique has been applied to instructions in [Mas+23; Hua+21] and to comments in *ReCode* [Wan+23]. This thesis additionally applies this perturbation to code, i.e., function names, variables, or constants. Since identifiers that contain multiple words are concatenated together, for the translation, those concatenation was joined by a space. After the translation, the space was then replaced by an underscore to form valid identifier names. **Backtranslation** is an easy way to include a paraphrasing perturbation in the framework. For the experiments of the thesis, the input is first translated into German and then back to English. Being a native German speaker, using German as the middle language was reasonable.

Butterfinger The **Butterfinger** perturbation introduces probability-based typos into the text. Specifically, this strategy assigns a random float value to each character of the input. If the assigned character value falls below a predefined threshold, the character is replaced with one of its nearest neighbors based on a *QWERTY*-keyboard layout [Wik25b].

Introducing typos is a relevant way to simulate real-world noise, which an LLM should be robust against. Typos are a common occurrence in real-world inputs, especially in code comments, variable names, or natural language instructions. Since minor errors do not change the intended semantics of a text, a robust model should not be misled by such a perturbation. The probability of a character being replaced was set to 0.05, as this value was also used and validated for naturalness in *ReCode* [Wan+23].

ChangeCharCase **ChangeCharCase** randomly changes the case of characters in the input. Similar to **Butterfinger**, a random float value is assigned to each character of the input. If the assigned value falls below a certain threshold, the character's case is flipped. This perturbation simulates real-world variations due to inconsistent casing conventions or accidental typing errors. While these changes don't affect the semantics of a text, they may affect the model's capabilities of processing the inputs due to the tokenization. Nonetheless, a robust LLM should be able to handle this without difference in performance.

This perturbation strategy has been previously applied to comments in *ReCode* [Wan+23]. This thesis goes beyond that and targets either instruction, comments, or code. The probability of a character switching its case was set to 0.3. *ReCode* validated the naturalness for a probability of 0.35, but early investigation showed that the perturbed prompts were changed too strongly. Therefore, a slightly lower probability was chosen in this thesis.

Concretizer The **Concretizer** perturbation is based on *COCO*'s perturbation [Yan+23a]. However, unlike *COCO*, which extracts features from the generated code, the code features in the thesis's perturbation are extracted from the original *C* code. Like *COCO*, this perturbation randomly generates either requiring or forbidding specifications in the instruction, based on the extracted features. The original *COCO* perturbation used a complex heuristic to select the concretizations added to the prompt. Since this perturbation is only one of many in this framework, it has been decided to simply randomly sample three to five feature constraints to make the implementation process more efficient.

Translation Similar to **Backtranslation**, the **Translation** perturbation is applied to instructions, comments, or code. This thesis enables assessing the robustness of a model when encountering inputs of different languages, which is a potential scenario in the real world. Many software development environments involve international teams, where documentation, comments, and even code identifiers may appear in multiple languages [LLC20; PJ15]. Evaluating how robustly models process varying languages is therefore beneficial in practice. Similar to **Backtranslation**, multi-word identifiers were split before translation and then joined by an underscore after being translated.

The perturbation translated the inputs into German and Korean. Choosing German follows the same reasoning as in **Backtranslation**, i.e, being a native speaker. In addition, Korean was selected based on prior language assessments of LLMs [Ahu+24]. This work shows that models generally perform well on Germanic languages (e.g., English or German), but their performance tends to degrade on languages that do not use Latin characters. Specifically, the paper highlights that "highly morphological languages" [Ahu+24] like Korean correlate with increased "tokenizer fertility"[1], which ultimately leads to worse

[1]Meaning that more tokens are required to represent the same text.

model performance.

LLMCommentInsertion This perturbation strategy prompts *GPT-4o-mini* [Ope24a] to add meaningful comments to the *C* code. Specifically, it first removes all existing comments using *tree-sitter* [Bru+25] and then prompts the LLM to insert comments. The prompt's instruction is:

> Enhance the clarity of the following C code by adding meaningful and relevant comments. Your comments should aim to help readers understand the code's functionality, logic, and structure. Feel free to use both multi-line and single-line comments as appropriate. Please ensure that you only add comments without altering the existing code.

Manually checking the perturbation results during development showed that this approach produces sufficiently relevant comments for this use case. However, relying on a LLM for perturbation introduces potential unpredictable results. To mitigate this, LLM-based perturbations included safety checks and *feedback loops*, which is briefly explained in subsection A.1.1. Nonetheless, the LLM-based process can not guarantee that comments are always relevant. However, the same limitation applies to comments written by human programmers, making this perturbation a realistic approach for evaluating robustness to differently documented code.

RemoveComments **RemoveComments** removes all comments from the *C* code, using *tree-sitter* [Bru+25]. The motivation behind this perturbation is that comments could serve as natural language explanations of the code, potentially helping the model understand the code's functionality. By removing comments, the model loses this additional information, which might result in deviating performance. This perturbation provides insight into the model's robustness when faced with poorly documented codebases.

CodeFormat The **CodeFormat** perturbation applies different code formatters, which produce different indentations and ensure that the code follows a consistent style. To format the code, the framework uses *clang-format* [LLVb] to perturb *C* code with either *Mozilla's* [Mozc] code style or *LLVM's* [LLVc] code style.

Coding conventions vary greatly in real-world environments and therefore, are relevant to be robust against. Code formatting does not change the semantics of the code, it only introduces structural modifications, which perfectly align with the requirements of perturbations.

ABC **ABC** is a code perturbation strategy that replaces all identifiers with a single-character representation in alphabetical order. If the number of unique identifiers exceeds the number of available characters, multi-character combinations are used. Such a renaming strategy does not change the functionality of the code, but significantly alters its readability. As a result, the perturbation helps to find out whether an LLM relies on meaningful identifier names for a successful translation.

CamelCase **CamelCase** applies the *CamelCase* notation to all identifiers. This means that multi-word identifiers are concatenated, with each word starting with a capital letter except for the first character. Similar to formatting styles, identifier casing conventions vary across codebases. While *Rust* has well-defined naming conventions [Thea], primarily relying on *snake_case*, *C* does not necessarily follow a universal standard. Instead, conventions can differ across projects [LLVc; GNUa], although there seems to be a move towards *snake_case* [µOS14].

With casing styles not being standardized across all projects, LLMs have to be robust against varying casing strategies. By converting all identifiers to *CamelCase*, this perturbation evaluates whether models are robust to such modifications. Understanding how different casing strategies impact the translation performance is relevant for practical applications, since models should be able to generalize across varying code conventions.

IdenObfuscator **IdenObfuscator** operates similarly to **ABC**, but instead of encoding identifiers into single characters, it removes entire words that form a complete word. For example, the identifier `calculate_mean` consists of two distinct words, and this perturbation would reduce it to `_`. By contrast, `calc_mean` contains only one complete word, which leads to **IdenObfuscator** producing `calc_`.

Since this approach can introduce duplicate identifiers, the system always verifies whether the updated identifier is unique. If not, the perturbation appends a number to the identifier, resulting in identifiers like `_1`, or `_12`.

The motivation behind this perturbation is similar to that of **ABC**. However, using only `_` with numbers may significantly reduce code readability, making it harder to understand. Obfuscated variables like these can occur in unintuitive automatic code generation. However, compared to other perturbation strategies, this approach has limited real-world applicability.

LLMVariableImprove This perturbation strategy utilizes a locally running *Llama-2-7b* [Tou+24] and tries to improve the identifier names. The original identifier is prompted to the model and instructed to generate an improved identifier. The instruction used for this prompt is as follows:

> You are a robot that shortens and improves variable names. You can only respond with one brief variable name and you cannot use the words Sure, Here or Okay. Use backticks around the variable name. For example, \`my_variable\`.

Identifier names can vary across codebases because of project guidelines, auto-generated code, or individual coding styles. Changing the variables name with this perturbation checks if an LLM generalizes well with variable names, which is a relevant capability for practical applications.

PascalCase **PascalCase** is the same as **CamelCase**, but the first letter is also capitalized. The perturbation's motivation follows that of **CamelCase**. In addition, **PascalCase** is less popular in practice than other casing styles. As a result, models may have encountered only a few times during training, which could potentially result in a more challenging casing convention for an LLM. Consequently, this perturbation helps to assess whether the model is robust to casing strategies that are less common in real-world codebases.

SnakeCase The **SnakeCase** perturbation applies *snake_case* to identifiers. Specifically, this means that multi-word identifiers are written in lowercase and separated by underscores.

The motivation for evaluating this casing convention follows that of **CamelCase**. Additionally, since *snake_case* is the standardized strategy in *Rust*, evaluating its impact could provide insights into whether using consistent casing strategies in code translation impacts model performance.

ConstantInsertion This perturbation was demonstrated in [CLS19] and adds unnecessary constant declarations to the *C* code. The perturbation uses a JSON [Bra17] file that contains random constant definitions. To apply the perturbation, the *C* code is parsed into an AST, and all possible positions for syntax-correct constant insertions are identified. Following that, the perturbation samples random constants from the JSON.

This perturbation only adds irrelevant declarations, which slightly change the code structure without affecting its logic. Considering that codebases evolve, irrelevant declarations are a potential input variation that could be encountered in practice, making robust performance desirable.

DeadCodeInsertion The **DeadCodeInsertion** perturbation, used in [Wan+23; Zha+23b], adds unused code snippets to the *C* code. Similar to **ConstantInsertion**, this perturbation utilizes a JSON [Bra17] file that holds a list of random code snippets.

Each code snippet object has a description and a `possible_insertions` list that provides where this snippet can be inserted. Since not all code snippets can be inserted everywhere[2], `possible_insertions` list ensures that each snippet is only inserted in syntax-correct positions. In detail, the perturbation parses the *C* code into an AST and characterizes the eligible insertion points, and then randomly samples an eligible code snippet for this insertion point from the JSON.

Also, the description for each inserted snippet can be added as a comment, which is controlled by a `comment_probability` parameter set to 50%. The perturbation additionally has an "max_snippets" parameter that is set to ten by default to avoid bloating of the code.

The perturbation is like **ConstantInsertion**, except the actual insertions are more complicated, making it harder for the model. It enables an assessment of how LLM-based code translation manages syntactically correct code that is not relevant for the functionality of the code. Since real-world code bases may have such unused code snippets due to unfinished implementations or legacy structures, it is worth evaluating how robust the model is in this situation for a real-world use.

IncludeCommentAdder This perturbation introduces additional include statements, or comments behind includes, depending on the configuration. While all other perturbations can be clearly distinguished as either targeting instruction, comment, or code, this perturbation always targets comments and code.

In order to explain this perturbation, it is best to present its configurable parameters.

- *used_calls*: If set to true, the perturbation adds comments that list the function names which are used in the code and originally come from the imported library. If set to false, the perturbation adds random function names of the important library that are not used in the code, which could potentially mislead the model.

[2]For example, functions cannot be defined inside other functions in *C*

- *add_includes*: If set to true, the perturbation ensures that all required includes are present for the *C* code.
- *add_random_includes*: This controls whether the perturbation adds random standard library [IBM25] includes. Given an integer, the perturbation randomly samples a number of includes from the standard library, and appends them with a comment.
- *comment_type*: Specifies whether the comments use // (double slash) or /*... */ (slash-asterisk).

In the context of real-world relevance, include statements that can sometimes be followed by comments, which explain why a library was imported. One way is by listing the functions that are used from the library. Due to outdated documentation or improper refactoring, these comments can also be incorrect, referencing functions that are no longer used.

Applying this perturbation enables the evaluation of whether LLMs are robust against potentially misleading import statements and comments. However, to enable a meaningful evaluation of this perturbation strategy, it has to ensure that the file contains include statements and add include statements in case the original code has none. This is necessary because the perturbation would otherwise not produce any change. This is the reason why this strategy does not strictly fit into the previously described categories of targeting either comments or code exclusively, as doing so would prevent its intended purpose.

LLMCodeExtraction The **LLMCodeExtraction** utilizes *GPT-4o-mini* to refactor the *C* code by extracting code blocks into separate functions. Specifically, it utilizes this prompt:

> You have to extract some code into functions in this C snippet. Keep the semantic meaning and dont make any changes except for extracting code into functions. Be careful not to introduce any side effects or syntax errors! Especially, when working with Exit Codes! You should rather leave the code as is than risk introducing side effects or syntax errors.

Since, this strategy involves a nondeterministic LLM, it uses the same safety mechanisms as other LLM-based perturbation strategies, as described in subsection A.1.1. Refactoring code by extracting blocks into separate functions is a common software engineering practice. This introduces structural changes that may affect how an LLM processes and understands the code.

In real-world scenarios, codebases frequently undergo refactoring. This perturbation evaluates whether an LLM can maintain performance when confronted with structurally different yet functionally equivalent code.

FunctionSignatureChange The **FunctionSignatureChange** perturbation modifies functions by adding random parameters to the function signature. Each call to the modified function is then updated accordingly by passing a value that matches the newly introduced parameter's type. Since the added parameters are never used inside the function preserves its semantic meaning, while introducing a structural transformation at Level IV.

In real-world codebases, unused function parameters can, for example, result from incomplete refactoring. While such parameters have no impact on the code's functionality, they change the structure of function definitions and calls. With this perturbation, the framework incorporates another perturbation to evaluate an LLM's ability to translate structurally modified code.

ForWhileSwitch The **ForWhileSwitch** converts *for-loops* into their equivalent *while-loop* representations and vice versa. This transformation is a well-known practice[3] for restructuring code while keeping semantic equivalence. This is another perturbation that changes the code's structure without modifying its behavior. However, perturbations utilizing such a semantic equivalent algorithmic tweak are a more profound change (Level V, see Figure 3.1) than only adding parameters to function signatures, for example By assessing robustness against major structural modifications, this perturbation provides insight into how well an LLM can generalize across different, but semantically equivalent, representations of control flow.

ConditionSwap **ConditionSwap** perturbation swaps operands and predicates in logical conditions, while ensuring semantic equivalence. To prevent unintended changes in program behavior, *logical AND* and *logical OR* predicates are not swapped, as this could interfere with *short-circuit evaluation* [Wik25c]. Table 4.1 shows that this perturbation is a well-established strategy for creating semantically equivalent code perturbations. This perturbation introduces syntactic modifications without changing the functionality while altering the conditions. Checking whether LLMs are robust against such transformations can reveal whether they have a robust understanding of logic.

ConditionDup The **ConditionDup** perturbation was introduced by Li et al. [Li+24b] and inserts logically irrelevant elements to conditional expressions (e.g., && *True* or || *False*). As described in section 3.3, these additions do not modify the semantics of the code but introduce structural changes.

Redundant conditional expressions can, for instance, arise due to auto-generated code. A robust model should not deviate in its performance for such logically irrelevant changes. Since **ConditionDup** explicitly modifies decision logic by extending conditions, it is categorized as a Level VI perturbation according to the levels of Faidhi and Robinson [FR87].

DeMorgan The **DeMorgan** perturbation introduces changes in decision logic while keeping semantical equivalence by definition. Specifically, it utilizes *De Morgan's Law* [Wik25a] on conditional statements:

$$a \land b \iff \neg(\neg a \lor \neg b)$$

$$a \lor b \iff \neg(\neg a \land \neg b)$$

Even though this perturbation may have less practical relevance than other perturbations, it leads to significant structural changes so that it serves as an effective test for evaluating an LLM's robustness against differently structured logical statements. This perturbation enables a rather simple implementable perturbation that causes deep modifications to the code in Level VI. Since deeper levels were underrepresented in related work, this strategy is beneficial for exploring the robustness to perturbations of deeper levels.

[3]See Table 4.1

A.2 Perturbation Configuration

A.2.1 Embedding of Deterministic Perturbations on Instructions

Figure A.1 shows that the deterministic perturbation strategies on instructions produce the same instruction for each file among the same perturbation strategy.

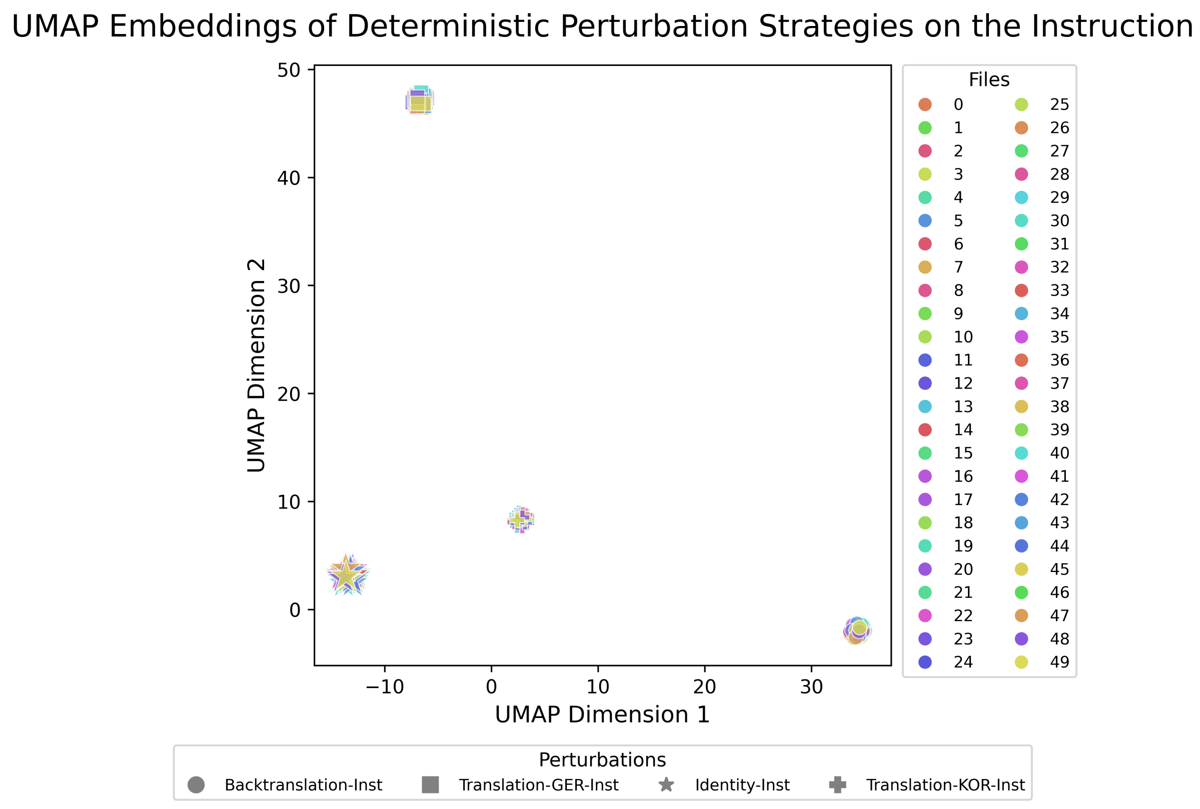


Figure A.1: UMAP embeddings of the 50 color-coded code files, each subjected to multiple **deterministic perturbation strategies** indicated by different markers. The plot shows that the different files still form clusters with their perturbations, suggesting that the perturbation strategies produced semantically similar **instructions**.

A.3 Baseline Performance for Different Parameters

Figure A.2 visualizes the impact of increasing s and k for $RP_s@k$ for *GPT-4o-mini*. It confirms that higher s results in higher difficulty for each k. Furthermore, higher k leads to improved results. Both findings are expected, as increasing s introduces the worst-case approach that is pruning variances, and larger k reduces sensitivity to variance, as it yields higher values for a less amount of correct k runs.

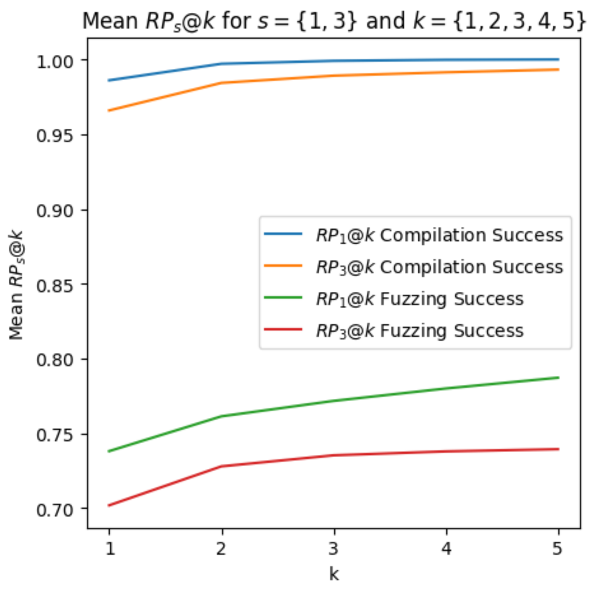


Figure A.2: Sampled $RP_s@k$ results across various k for $s = 1$ and $s = 3$. Detailing higher values for increasing k.

A.4 Error Rates per Model

This section gives a small overview of the observed error rates.

A.4.1 GPT-4o-mini

Figure A.3 details the aggregated error rates for all deterministic perturbation strategies as a percentage. These rates are almost similar to the error rates of the baseline in Figure 5.6. Files 18 and 43 showing significant rates for either *fuzzing exception* or *fuzzing setup*. Both of these error types can be caused by the model not strictly following the intent and maybe producing different function names, or problematic functions, which the fuzzer can not call. Considering these errors stem from incorrect translations, those of the *translation system* or *LLM API* would affect the interpretability more. However, these error rates are rather low.

Figure A.4 shows that **IdenObfuscator** produced the most errors in the *translation system*, yet in only 1.2% of calls. Recall that **IdenObfuscator** produced the worst results and failed for the previously perfect or variant files: 7, 19, 41, 42, and 44. Figure A.3 shows that none of these files were affected by errors of the *translation system* or *LLM API*. Instead, the **IdenObfuscator** seemingly produced different function names, leading to 10% error rate in the *fuzzing setup*. This effect is primarily due to the perturbation

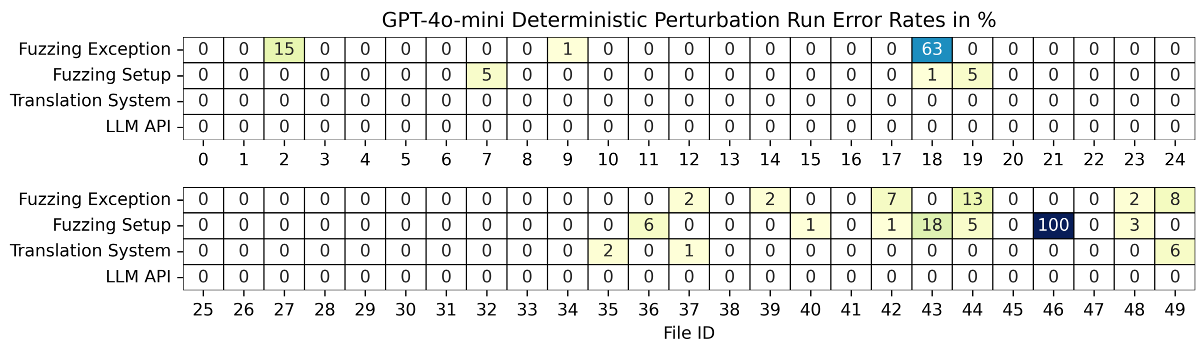


Figure A.3: Errors rates per file in % aggregated for all deterministic perturbation strategies for *GPT-4o-mini*.

sometimes producing the function name "_", which is syntactically possible in *C* but not in *Rust*, which therefore makes a compilable *Rust* version with identical function names impossible. However, as detailed, this was not the primary reason for **IdenObfuscator** causing non-robust performance, as this only happened to files that could also not be translated successfully for the **Identity** version.

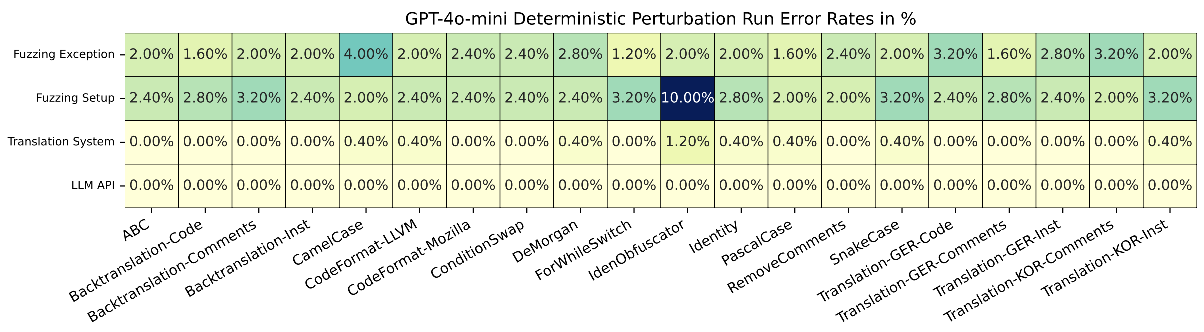


Figure A.4: Error rates per deterministic perturbation in % for *GPT-4o-mini*.

A.4.2 Other Models

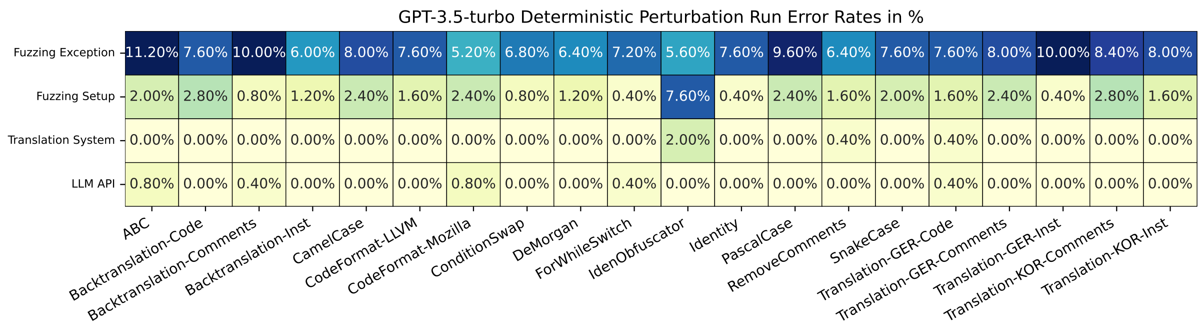
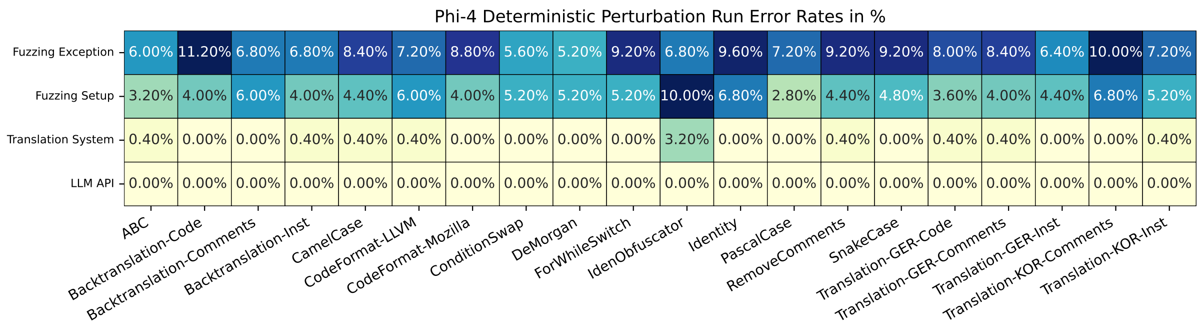
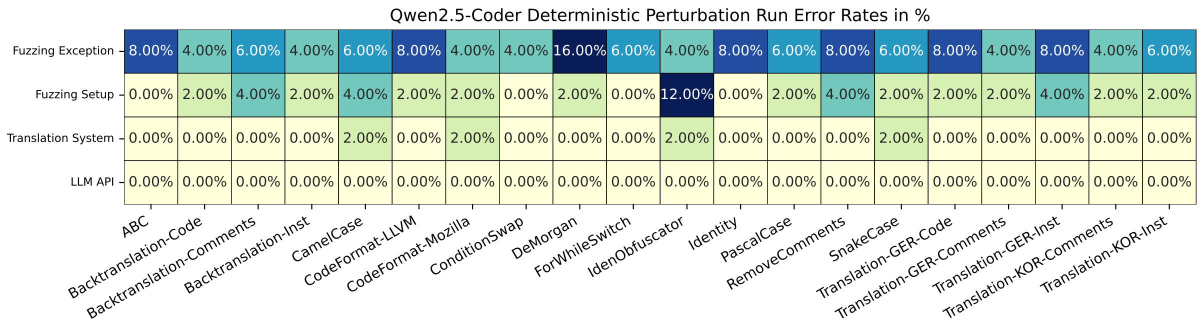


Figure A.5: Error rates per deterministic perturbation in % for *GPT-3.5-turbo*.



	ABC	Backtranslation-Code	Backtranslation-Comments	Backtranslation-Inst	CamelCase	CodeFormat-LLVM	CodeFormat-Mozilla	ConditionSwap	DeMorgan	ForWhileSwitch	IdenObfuscator	Identity	PascalCase	RemoveComments	SnakeCase	Translation-GER-Code	Translation-GER-Comments	Translation-GER-Inst	Translation-KOR-Comments	Translation-KOR-Inst
Fuzzing Exception	6.00%	11.20%	6.80%	6.80%	8.40%	7.20%	8.80%	5.60%	5.20%	9.20%	6.80%	9.60%	7.20%	9.20%	9.20%	8.00%	8.40%	6.40%	10.00%	7.20%
Fuzzing Setup	3.20%	4.00%	6.00%	4.00%	4.40%	6.00%	4.00%	5.20%	5.20%	5.20%	10.00%	6.80%	2.80%	4.40%	4.80%	3.60%	4.00%	4.40%	6.80%	5.20%
Translation System	0.40%	0.00%	0.00%	0.40%	0.40%	0.40%	0.00%	0.00%	0.00%	0.00%	3.20%	0.00%	0.00%	0.40%	0.00%	0.40%	0.40%	0.00%	0.00%	0.40%
LLM API	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Figure A.6: Error rates per deterministic perturbation in % for *Phi-4*.



	ABC	Backtranslation-Code	Backtranslation-Comments	Backtranslation-Inst	CamelCase	CodeFormat-LLVM	CodeFormat-Mozilla	ConditionSwap	DeMorgan	ForWhileSwitch	IdenObfuscator	Identity	PascalCase	RemoveComments	SnakeCase	Translation-GER-Code	Translation-GER-Comments	Translation-GER-Inst	Translation-KOR-Comments	Translation-KOR-Inst
Fuzzing Exception	8.00%	4.00%	6.00%	4.00%	6.00%	8.00%	4.00%	4.00%	16.00%	6.00%	4.00%	8.00%	6.00%	8.00%	6.00%	8.00%	4.00%	8.00%	4.00%	6.00%
Fuzzing Setup	0.00%	2.00%	4.00%	2.00%	4.00%	2.00%	2.00%	0.00%	2.00%	0.00%	12.00%	0.00%	2.00%	4.00%	2.00%	2.00%	2.00%	4.00%	2.00%	2.00%
Translation System	0.00%	0.00%	0.00%	0.00%	2.00%	0.00%	2.00%	0.00%	0.00%	0.00%	2.00%	0.00%	0.00%	0.00%	2.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LLM API	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Figure A.7: Error rates per deterministic perturbation in % for *Qwen2.5-Coder*.

A.5 Correlation of Cosine Similarity and Translation Success

A.5.1 Compilation Success for GPT-4o-mini

Deterministic Perturbations

Figure A.8 shows the *cosine similarity* and *GPT-4o-mini*'s *compilation success* under *deterministic perturbations*. Without *feedback loops* there is a correlation of −0.166 (weak) and with *feedback loops* it is 0.013 (weak). Consequently, there is no correlation between *cosine similarity* and *compilation success* for *deterministic perturbations* on code.

Figure A.9 visualizes the same for perturbations on instructions. The correlation coefficient between *cosine similarity* and $RC_1@5$ could not be calculated for this data, because the $RC_1@5$ had the same value for every perturbation. That means $RC_1@5$ does not show any variance, so its standard deviation is zero. Since calculating the correlation coefficient involves dividing by the standard deviation, this results in an undefined operation.

Stochastic Perturbations

Figure A.8 visualizes the correlation between *cosine similarity* and *compilation success* for *GPT-4o-mini* under *stochastic perturbations*. Without *feedback loops* there is a correlation of −0.738 (strong) and with *feedback loops* −0.214 (weak). That shows again that *feedback loops* might improve robustness for less similar inputs.

Figure A.11 illustrates the *stochastic perturbations* on instructions. Similar to *fuzzing success*, the correlation coefficient yields 1.0, as there are only two data points, making this assessment not meaningful.

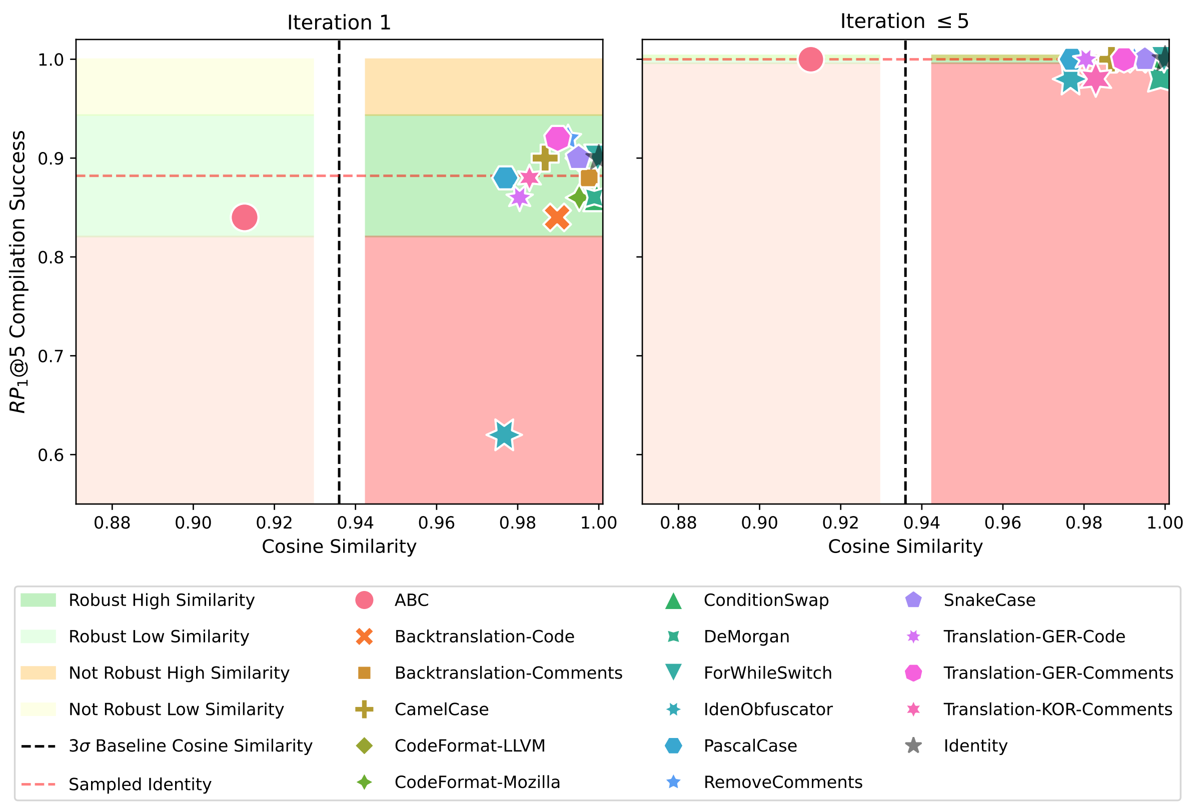


Figure A.8: *Cosine similarity* and *compilation success* of *deterministic code perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

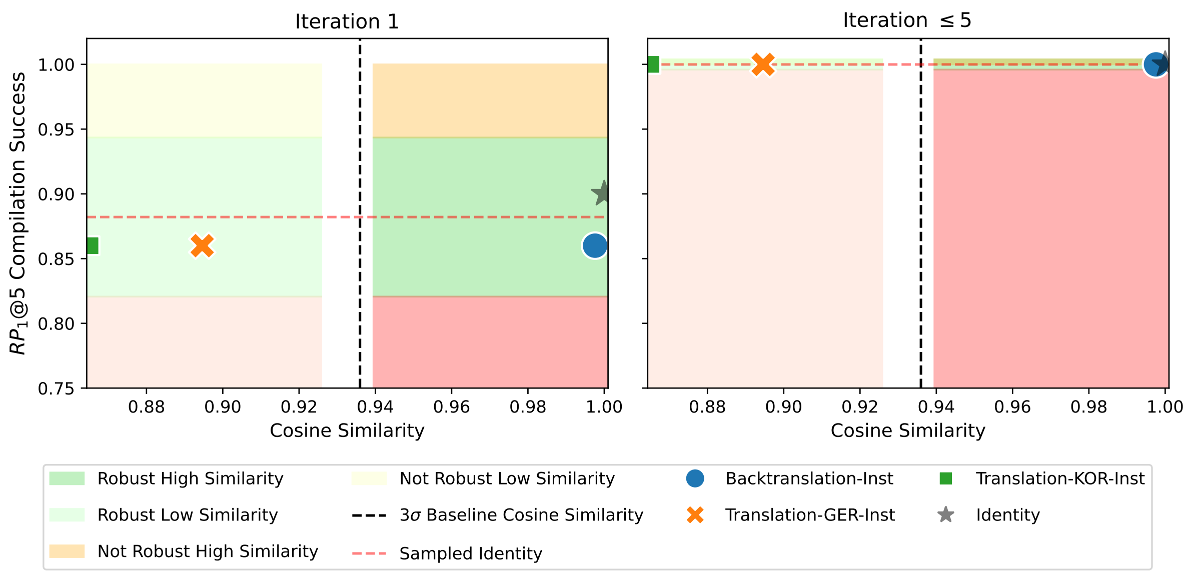


Figure A.9: *Cosine similarity* and *compilation success* of *deterministic instruction perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

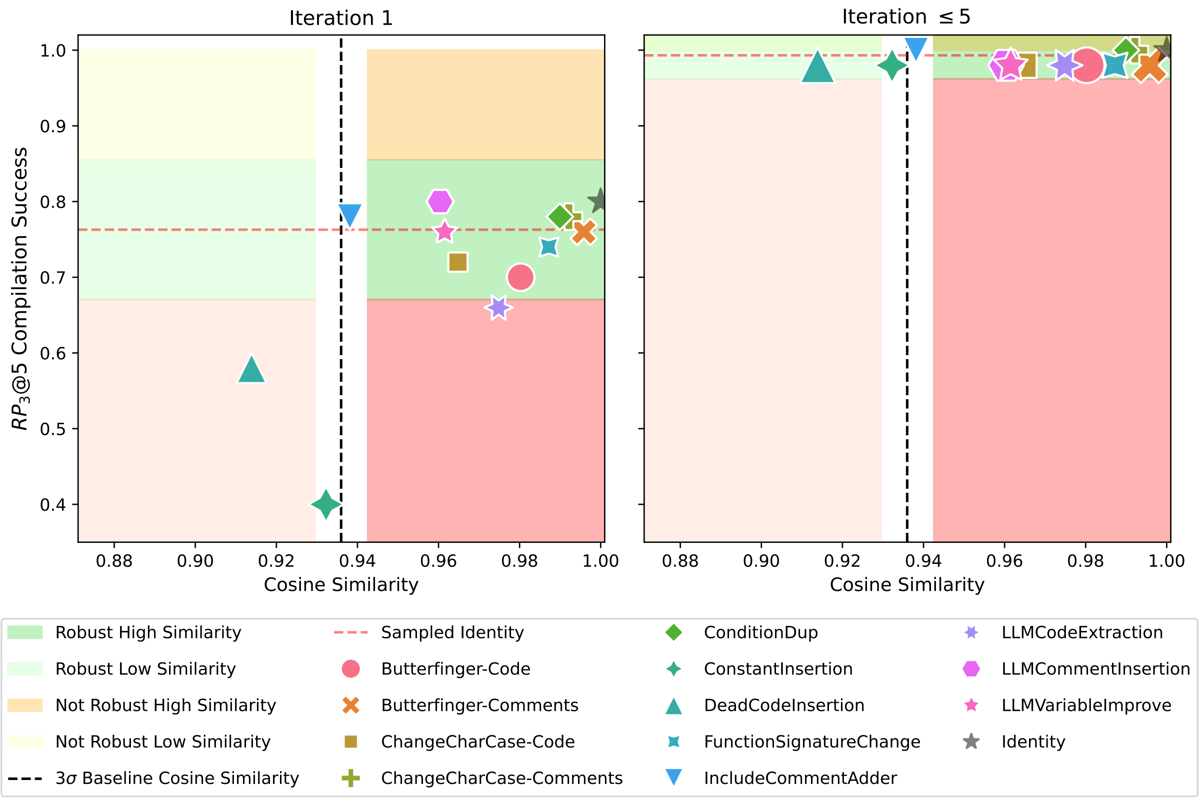


Figure A.10: *Cosine similarity* and *compilation success* of *stochastic code perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

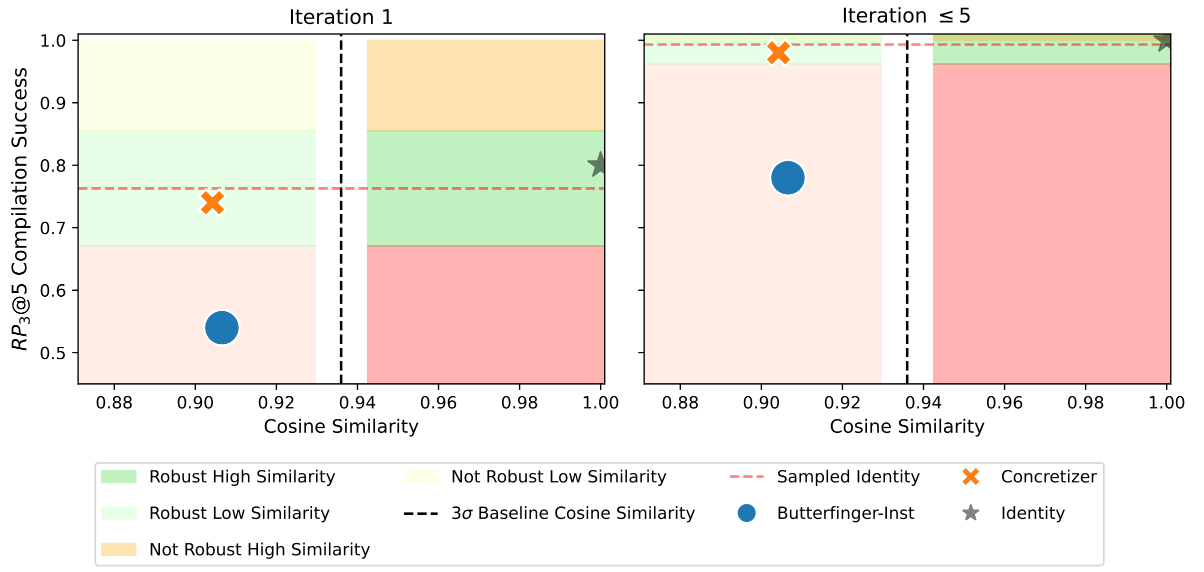


Figure A.11: *Cosine similarity* and *compilation success* of *stochastic instruction perturbations*. Divided into perturbations with high and low similarity, and robust or non-robust behavior.

A.6 Overview of Used Tools

A.6.1 Artificial Intelligence Tools

This section details the AI-tools that were used during the creation of this thesis. Table A.2 lists these used tools and explains for what tasks they were used and how. As LLMs are a main component of this thesis, they do not need an extra explanation. In addition, the thesis sparsely tested *OpenAI's* new *image generation model* [Ope25a] and experimented with *eraser*[4]. The exact generations of these visual tools did not make it into the thesis, but sparked ideas for creativity, which were then utilized manually with the design tool *Figma*[5]. An intermediate step was to vectorize the generated images with *Recraft AI*[6] to work with them in *Figma*. Furthermore, for consistency and readability, *Grammarly*[7] was used. Phrases or words that went beyond my English vocabulary were translated with *Google Translate*[8] or *DeepL*[9]. Lastly, *Github Copilot*[10] is a commonly used tool in *Bosch's* development environment. The thesis benefited from its capabilities in auto-completion, inline edits, and chat features. While the thesis's readability, creative examples in visualizations, or coding efficiency might have benefited from using these tools, I can confidently ensure that the "scientific contribution" of this work was entirely made by myself.

A.6.2 Online Tools

To complete the overview of used tools, Table A.3 lists notable online tools, that have been used for the creation of this thesis.

[4]https://www.eraser.io/
[5]https://www.figma.com/
[6]https://www.recraft.ai/
[7]https://grammarly.com/
[8]https://translate.google.com/
[9]https://www.deepl.com/
[10]https://github.com/features/copilot
[11]https://www.bosch-presse.de/pressportal/de/en/generative-ai-at-bosch-263243.html
[12]This image was not good, but sparked the idea for the bowl with green and red balls.
[13]https://scholar.google.de/
[14]https://dblp.org/
[15]https://www.powerthesaurus.org/
[16]https://www.figma.com
[17]https://gpt-tokenizer.dev

Tool	Use Case	Used Prompts/Examples
AskBosch[11] (GPT-4o-mini) AskBosch (GPT-4o-mini)	Fine-tuning the wording of the thesis's title Conciseness of chapter titles, section titles or captions	"I want to improve the title of my master thesis. What is a good title that incorporates 'development'?" "Is there a way of making the figure caption more crisp: . . ."
AskBosch (GPT-4o-mini) ChatGPT (o3-mini)	Latex table **templating**.	"Can you flip the table so that it makes more sense?", "Turn this csv into a latex table that is not too wide horizontally. Use LineBreaks so that everything is visible on a vertical A4 sheet."
ChatGPT (4o with search)	Finding "best practice" references	"Is there an original source for cosine similarity between embedding vectors that is commonly cited?"
ChatGPT (4o-mini) AskBosch (GPT-4o-mini) ChatGPT(4o)	Find common English terms Correcting Grammar	"How do you say to 'the best of your knowledge' if you don't want to use 'our'" "Is this correct English?: . . ."
ChatGPT (4o) ChatGPT (Image Generation) Recraft AI (Vectorizing) Eraser AI	A creative example for illustrating tokenization in Figure 2.1 Initial idea of Figure 2.2. The used figure only uses an AI-generated bowl and the hand holding a ball. The single vectors were set together in Figma by hand. Was initially used to try generating the framework overview (Figure 4.1). However, since no generation was satisfactory, this was discarded, and the visualizations were made with Figma. It is still mentioned because the general look of the figures was inspired by the initial attempt.	"I need a creative example for tokenization visualization, which fits to a thesis with the title . . ." "Pass@k is defined like this . . . Can you come up with a great picture based on this, so that one directly understands the intuition behind pass@k?"[12]"Just generate me an empty bowl." "I need an architecture diagram for a framework with three components . . ."
Grammarly	Correction of grammatical mistakes and phrasing.	
DeepL GoogleTranslator	Translation of words or phrases.	German to English
Github Copilot	Auto-completion, figure generation with the chat-feature, example code for Appendix A.1, and prompt refinements for LLM-based perturbations.	"How to include the *Z-Score* value for each bar . . .", "Make the following LLM prompt more concise: . . ."

Table A.2: Overview of the used AI-tools.

Tool	Use Case
Google Scholar[13] DBLP[14]	Getting curated and consistent BibTeX entries for found papers from the literature search.
PowerThesaurus[15]	Find synonyms to prevent repetition and extend the vocabulary.
Figma[16]	Design Tool used for the Figures 2.2, 2.3, 4.1, 4.2, and 4.3.
GPT-Tokenizer Playground[17]	Created the visualization of the tokenization example in Figure 2.1.

Table A.3: Overview of used online tools.

Bibliography

[Aba+24] Asma Ben Abacha, Wen-wai Yim, Yujuan Fu, Zhaoyi Sun, Meliha Yetisgen, Fei Xia, and Thomas Lin. “MEDEC: A Benchmark for Medical Error Detection and Correction in Clinical Notes”. In: *CoRR* abs/2412.19260 (2024). DOI: 10.48550/ARXIV.2412.19260 (cit. on p. 66).

[Abd+24] Marah I Abdin, Jyoti Aneja, Harkirat S. Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. “Phi-4 Technical Report”. In: *CoRR* abs/2412.08905 (2024). DOI: 10.48550/ARXIV.2412.08905 (cit. on pp. 7, 14, 18, 66, 113).

[AHS85] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. “A Learning Algorithm for Boltzmann Machines”. In: *Cogn. Sci.* 9.1 (1985), pp. 147–169. DOI: 10.1207/S15516709COG0901_7 (cit. on p. 15).

[Aga+24] Anisha Agarwal, Aaron Chan, Shubham Chandel, Jinu Jang, Shaun Miller, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Neel Sundaresan, and Michele Tufano. “Copilot Evaluation Harness: Evaluating LLM-Guided Software Programming”. In: *arXiv preprint arXiv:2402.14261* (2024). arXiv: 2402.14261 (cit. on pp. 5, 17, 36).

[Aho+14] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. “A Simple Syntax-Directed Translator”. In: *Compilers: Principles, Techniques, and Tools*. Second edition, Pearson new international edition. Always Learning. Harlow: Pearson, 2014, pp. 39–107. ISBN: 978-1-292-02434-9 (cit. on p. 11).

[Ahu+24] Sanchit Ahuja, Divyanshu Aggarwal, Varun Gumma, Ishaan Watts, Ashutosh Sathe, Millicent Ochieng, Rishav Hada, Prachi Jain, Mohamed Ahmed, Kalika Bali, and Sunayana Sitaram. “MEGAVERSE: Benchmarking Large Language Models across Languages, Modalities, Models and Tasks”. In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*. Association for Computational Linguistics, 2024, pp. 2598–2637. DOI: 10.18653/V1/2024.NAACL-LONG.143 (cit. on pp. 26, 156).

[AG24] Jay Alammar and Maarten Grootendorst. “An Introduction to Large Language Models”. In: *Hands-On Large Language Models: Language Understanding and Generation*. First edition. Beijing Boston Farnham: O’Reilly, 2024. ISBN: 978-1-09-815096-9 (cit. on p. 14).

[Als+24] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. “Automated Unit Test Improvement Using Large Language Models at Meta”. In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. ACM, 2024, pp. 185–196. DOI: 10.1145/3663529.3663839 (cit. on p. 2).

[AHP18] Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. “The Meaning of Memory Safety”. In: *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Vol. 10804. Lecture Notes in Computer Science. Springer, 2018, pp. 79–105. DOI: 10.1007/978-3-319-89722-6_4 (cit. on pp. 9, 10).

[Ani+23] Rohan Anil et al. “Gemini: A Family of Highly Capable Multimodal Models”. In: *CoRR* abs/2312.11805 (2023). DOI: 10.48550/ARXIV.2312.11805 (cit. on p. 1).

[ANA23] Owura Asare, Meiyappan Nagappan, and N. Asokan. “Is GitHub’s Copilot as Bad as Humans at Introducing Vulnerabilities in Code?” In: *Empirical Software Engineering* 28.6 (2023), p. 129. DOI: 10.1007/S10664-023-10380-1 (cit. on pp. 1, 16).

[Aus+21] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. “Program Synthesis with Large Language Models”. In: *CoRR* abs/2108.07732 (2021). arXiv: 2108.07732 (cit. on pp. 17, 18, 27).

[BBH18] Roberto Bagnara, Abramo Bagnara, and Patricia M. Hill. “The MISRA C Coding Standard and Its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software”. In: *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*. Vol. 11002. Lecture Notes in Computer Science. Springer, 2018, pp. 5–23. DOI: 10.1007/978-3-319-99725-4_2 (cit. on p. 2).

[Ben+03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. “A Neural Probabilistic Language Model”. In: *J. Mach. Learn. Res.* 3 (2003), pp. 1137–1155. URL: https://jmlr.org/papers/v3/bengio03a.html (cit. on p. 13).

[BLP22] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. “Provably-Safe Multilingual Software Sandboxing Using WebAssembly”. In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 2022, pp. 1975–1992. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya (cit. on p. 23).

[Bou+22] Nicholas Boucher, Ilia Shumailov, Ross Anderson, and Nicolas Papernot. “Bad Characters: Imperceptible NLP Attacks”. In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 1987–2004. DOI: 10.1109/SP46214.2022.9833641 (cit. on p. 32).

[Bra17] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format.* Dec. 2017. DOI: 10.17487/RFC8259 (cit. on pp. 22, 159).

[Bro+20] Tom B. Brown et al. “Language Models Are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, Virtual.* 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html (cit. on pp. 14, 30).

[Cai+25] Xuemeng Cai, Jiakun Liu, Xiping Huang, Yijun Yu, Haitao Wu, Chunmiao Li, Bo Wang, Imam Nur Bani Yusuf, and Lingxiao Jiang. “RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM”. In: *CoRR* abs/2503.17741 (2025). DOI: 10.48550/ARXIV.2503.17741 (cit. on pp. 11, 65).

[Cha+23] Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan. “Transformer-Based Vulnerability Detection in Code at EditTime: Zero-shot, Few-shot, or Fine-tuning?” In: *CoRR* abs/2306.01754 (2023). DOI: 10.48550/ARXIV.2306.01754 (cit. on p. 18).

[CLS19] Hayden Cheers, Yuqing Lin, and Shamus P. Smith. “SPPlagiarise: A Tool for Generating Simulated Semantics-Preserving Plagiarism of Java Source Code”. In: *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS).* Beijing, China: IEEE, Oct. 2019, pp. 617–622. ISBN: 978-1-72810-945-9. DOI: 10.1109/ICSESS47205.2019.9040853. (Visited on 12/17/2024) (cit. on pp. 33, 45, 159).

[Che+24] Chao Chen, Kai Liu, Ze Chen, Yi Gu, Yue Wu, Mingyuan Tao, Zhihang Fu, and Jieping Ye. “INSIDE: LLMs’ Internal States Retain the Power of Hallucination Detection”. In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024.* OpenReview.net, 2024. URL: https://openreview.net/forum?id=Zj12nzlQbz (cit. on p. 58).

[Che+21] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374 (cit. on pp. 15, 17, 18, 27, 30, 53).

[Che+20] Minhao Cheng, Jinfeng Yi, Pin-Yu Chen, Huan Zhang, and Cho-Jui Hsieh. “Seq2Sick: Evaluating the Robustness of Sequence-to-Sequence Models with Adversarial Examples”. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020.* AAAI Press, 2020, pp. 3601–3608. DOI: 10.1609/AAAI.V34I04.5767 (cit. on p. 32).

[Chi19] Shigeru Chiba. “Foreign Language Interfaces by Code Migration”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019.* ACM, 2019, pp. 1–13. DOI: 10.1145/3357765.3359521 (cit. on p. 22).

[DG06] Jesse Davis and Mark Goadrich. "The Relationship between Precision-Recall and ROC Curves". In: *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*. Vol. 148. ACM International Conference Proceeding Series. ACM, 2006, pp. 233–240. DOI: 10.1145/1143844.1143874 (cit. on p. 60).

[Det+23] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. "QLoRA: Efficient Finetuning of Quantized LLMs". In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. 2023. URL: http://papers.nips.cc/paper%5C_files/paper/2023/hash/1feb87871436031bdc0f2beaa62a049b-Abstract-Conference.html (cit. on p. 15).

[Din+23] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. "Large Language Models of Code Fail at Completing Code with Potential Bugs". In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. 2023. URL: http://papers.nips.cc/paper%5C_files/paper/2023/hash/819cebb05f993840e8a52d7564c5c282-Abstract-Conference.html (cit. on pp. 16, 21).

[Dou+23] Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. "Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey". In: *CoRR* abs/2308.01191 (2023). DOI: 10.48550/ARXIV.2308.01191 (cit. on p. 33).

[Du+24] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. "Evaluating Large Language Models in Class-Level Code Generation". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, 81:1–81:13. DOI: 10.1145/3597503.3639219 (cit. on pp. 18, 19).

[DSL25] Yali Du, Hui Sun, and Ming Li. *Post-Incorporating Code Structural Knowledge into LLMs via In-Context Learning for Code Translation*. Mar. 2025. arXiv: 2503.22776 `[cs]`. (Visited on 04/21/2025) (cit. on p. 20).

[ET94] Bradley Efron and Robert J Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall/CRC, 1994. DOI: https://doi.org/10.1201/9780429246593 (cit. on p. 61).

[Elm90] Jeffrey L. Elman. "Finding Structure in Time". In: *Cogn. Sci.* 14.2 (1990), pp. 179–211. DOI: 10.1207/S15516709COG1402_1 (cit. on p. 14).

[Emr+21] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. "Translating C to Safer Rust". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–29. DOI: 10.1145/3485498 (cit. on pp. 2, 11).

[Eni+24] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. “Towards Translating Real-World Code with LLMs: A Study of Translating to Rust”. In: *CoRR* abs/2405.11514 (2024). DOI: 10.48550/ARXIV.2405.11514 (cit. on pp. 2, 19, 21–23, 40, 52, 64, 72, 140).

[FR87] J.A.W. Faidhi and S.K. Robinson. “An Empirical Approach for Detecting Program Similarity and Plagiarism within a University Programming Environment”. In: *Computers & Education* 11.1 (Jan. 1987), pp. 11–19. ISSN: 03601315. DOI: 10.1016/0360-1315(87)90042-X. (Visited on 01/21/2025) (cit. on pp. 33, 34, 37, 44, 136, 161).

[Fan+23] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. “Large Language Models for Software Engineering: Survey and Open Problems”. In: *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 31–53. DOI: 10.1109/ICSE-FOSE59343.2023.00008 (cit. on pp. 1, 9, 15–17).

[FLD18] Angela Fan, Mike Lewis, and Yann N. Dauphin. “Hierarchical Neural Story Generation”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics, 2018, pp. 889–898. DOI: 10.18653/V1/P18-1082 (cit. on p. 15).

[Fri+23] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. “InCoder: A Generative Model for Code Infilling and Synthesis”. In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/forum?id=hQwb-lbM6EL (cit. on p. 29).

[Gan+24] Shubham Gandhi, Manasi Patwardhan, Jyotsana Khatri, Lovekesh Vig, and Raveendra Kumar Medicherla. “Translation of Low-Resource COBOL to Logically Correct and Readable Java Leveraging High-Resource Java Refinement”. In: *LLM4CODE@ICSE*. 2024, pp. 46–53. DOI: 10.1145/3643795.3648388 (cit. on p. 149).

[GG24] Carlos Adriano Gonçalves and Célia Talma Gonçalves. “Assessment on the Effectiveness of GitHub Copilot as a Code Assistance Tool: An Empirical Study”. In: *Progress in Artificial Intelligence - 23rd EPIA Conference on Artificial Intelligence, EPIA 2024, Viana Do Castelo, Portugal, September 3-6, 2024, Proceedings, Part III*. Vol. 14969. Lecture Notes in Computer Science. Springer, 2024, pp. 27–38. DOI: 10.1007/978-3-031-73503-5_3 (cit. on p. 16).

[Gui+24] Giovani Guizzo, Jie M. Zhang, Federica Sarro, Christoph Treude, and Mark Harman. “Mutation Analysis for Evaluating Code Translation”. In: *Empir. Softw. Eng.* 29.1 (2024), p. 19. DOI: 10.1007/S10664-023-10385-W (cit. on p. 11).

[Guo+18] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. “DLFuzz: Differential Fuzzing Testing of Deep Learning Systems”. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 739–743. DOI: 10.1145/3236024.3264835 (cit. on p. 21).

[Guo+24] Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. “CodeEditorBench: Evaluating Code Editing Capability of Large Language Models”. In: *CoRR* abs/2404.03543 (2024). DOI: 10.48550/ARXIV.2404.03543 (cit. on p. 19).

[HQS24] Jesko Hecking-Harbusch, Jochen Quante, and Maximilian Schlund. “Formal Runtime Error Detection During Development in the Automotive Industry”. In: *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part I*. Vol. 14499. Lecture Notes in Computer Science. Springer, 2024, pp. 3–26. DOI: 10.1007/978-3-031-50524-9_1 (cit. on p. 63).

[Hen+21] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. “Measuring Coding Challenge Competence With APPS”. In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, Virtual*. 2021. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html (cit. on pp. 17, 18).

[Hin+16] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. “On the Naturalness of Software”. In: *Commun. ACM* 59.5 (2016), pp. 122–131. DOI: 10.1145/2902362 (cit. on p. 15).

[Ha98] Jerry L. Hintze and Ray D. Nelson and. “Violin Plots: A Box Plot-Density Trace Synergism”. In: *The American Statistician* 52.2 (1998), pp. 181–184. DOI: 10.1080/00031305.1998.10480559 (cit. on p. 120).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/NECO.1997.9.8.1735 (cit. on p. 14).

[Hol+20] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. “The Curious Case of Neural Text Degeneration”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=rygGQyrFvH (cit. on pp. 2, 5, 15, 29).

[HR25] Jaemin Hong and Sukyoung Ryu. “Type-Migrating C-to-Rust Translation Using a Large Language Model”. In: *Empir. Softw. Eng.* 30.1 (2025), p. 3. DOI: 10.1007/S10664-024-10573-2 (cit. on p. 2).

[HJ24] Wenpin Hou and Zhicheng Ji. “A Systematic Evaluation of Large Language Models for Generating Programming Code”. In: *CoRR* abs/2403.00894 (2024). DOI: 10.48550/ARXIV.2403.00894 (cit. on p. 5).

[Hou+24] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. “Large Language Models for Software Engineering: A Systematic Literature Review”. In: *ACM Trans. Softw. Eng. Methodol.* 33.8 (2024), 220:1–220:79. DOI: 10.1145/3695988 (cit. on p. 1).

[Hu+22] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. “LoRA: Low-Rank Adaptation of Large Language Models”. In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=nZeVKeeFYf9 (cit. on p. 15).

[Hua+21] Shuo Huang, Zhuang Li, Lizhen Qu, and Lei Pan. “On Robustness of Neural Semantic Parsers”. In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, April 19 - 23, 2021*. Association for Computational Linguistics, 2021, pp. 3333–3342. DOI: 10.18653/V1/2021.EACL-MAIN.292 (cit. on pp. 31, 45, 155).

[Hui+24] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. “Qwen2.5-Coder Technical Report”. In: *CoRR* abs/2409.12186 (2024). DOI: 10.48550/ARXIV.2409.12186 (cit. on pp. 7, 18, 66).

[IEE90] IEEE. “IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064 (cit. on p. 9).

[IEE24] IEEE. “IEEE Standard for Robustness Evaluation Test Methods for a Natural Language Processing Service That Uses Machine Learning”. In: *IEEE Std 3168-2024* (2024), pp. 1–29. DOI: 10.1109/IEEESTD.2024.10631891 (cit. on p. 25).

[IN16] Eugene Ilyushin and Dmitry Namiot. “On Source-to-Source Compilers”. In: *International Journal of Open Information Technologies* 4 (Apr. 2016) (cit. on p. 11).

[Imp+25] Cristina Improta, Pietro Liguori, Roberto Natella, Bojan Cukic, and Domenico Cotroneo. “Enhancing Robustness of AI Offensive Code Generators via Data Augmentation”. In: *Empirical Software Engineering* 30.1 (Jan. 2025), p. 7. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-024-10569-y. (Visited on 01/30/2025) (cit. on pp. 3, 5, 31, 35, 46, 136, 151).

[ISO24] ISO (the International Organization for Standardization). *Road Vehicles — Safety and Artificial Intelligence*. Tech. rep. First edition. Dec. 2024 (cit. on p. 26).

[IC21] ISO (the International Organization for Standardization) and IEC (the International Electrotechnical and Commission). *Artificial Intelligence (AI) — Assessment of the Robustness of Neural Networks — Part 1: Overview*. Tech. rep. First edition. Mar. 2021 (cit. on p. 25).

[ISO17] ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission), IEEE. *Systems and Software Engineering - Vocabulary*. Tech. rep. Sept. 2017 (cit. on p. 25).

[IC22] ISO (the International Organization for Standardization) and IEC (the International Electrotechnical) and Commission. *Information Technology — Artificial Intelligence — Artificial Intelligence Concepts and Terminology*. Tech. rep. First edition. July 2022 (cit. on p. 25).

[Jan+24] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. "CoTran: An LLM-Based Code Translator Using Reinforcement Learning with Feedback from Compiler and Symbolic Execution". In: *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*. Vol. 392. Frontiers in Artificial Intelligence and Applications. IOS Press, 2024, pp. 4011–4018. DOI: 10.3233/FAIA240968 (cit. on p. 21).

[Jes+23] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. "Large Language Models and Simple, Stupid Bugs". In: *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 2023, pp. 563–575. DOI: 10.1109/MSR59073.2023.00082 (cit. on pp. 1, 16).

[Jia+23] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. "Impact of Code Language Models on Automated Program Repair". In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1430–1442. DOI: 10.1109/ICSE48619.2023.00125 (cit. on p. 16).

[Jun+18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154 (cit. on pp. 2, 10).

[JM25] Daniel Jurafsky and James H. Martin. "Neural Networks". In: *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. 2025. URL: https://web.stanford.edu/~jurafsky/slp3/ (cit. on p. 14).

[Kap+20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. "Scaling Laws for Neural Language Models". In: *CoRR* abs/2001.08361 (2020). arXiv: 2001.08361 (cit. on p. 14).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a Meeting Held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 2012, pp. 1106–1114. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (cit. on p. 33).

[Lat+23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Verus: Verifying Rust Programs Using Linear Ghost Types". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 286–315. DOI: 10.1145/3586037 (cit. on p. 23).

[Lem+23] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pretrained Large Language Models". In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 2023, pp. 919–931. DOI: 10.1109/ICSE48619.2023.00085 (cit. on p. 16).

[Lev66] Vladimir I Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals". In: *Soviet Physics Doklady.* Vol. 10. Soviet Union, 1966, pp. 707–710 (cit. on pp. 30, 56, 57).

[LH13] Baoli Li and Liping Han. "Distance Weighted Cosine Similarity Measure for Text Classification". In: *Intelligent Data Engineering and Automated Learning - IDEAL 2013 - 14th International Conference, IDEAL 2013, Hefei, China, October 20-23, 2013. Proceedings.* Vol. 8206. Lecture Notes in Computer Science. Springer, 2013, pp. 611–618. DOI: 10.1007/978-3-642-41278-3_74 (cit. on p. 13).

[LM24] Daniel Li and Lincoln Murr. "HumanEval on Latest GPT Models - 2024". In: *CoRR* abs/2402.14852 (2024). DOI: 10.48550/ARXIV.2402.14852 (cit. on p. 5).

[Li+19] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. "TextBugger: Generating Adversarial Text Against Real-world Applications". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/textbugger-generating-adversarial-text-against-real-world-applications/ (cit. on p. 32).

[Li+23] Raymond Li et al. "StarCoder: May the Source Be with You!" In: *Trans. Mach. Learn. Res.* 2023 (2023). URL: https://openreview.net/forum?id=KoFOg41haE (cit. on p. 18).

[Li+25] Ruishi Li, Bo Wang, Tianyu Li, Prateek Saxena, and Ashish Kundu. "Translating C To Rust: Lessons from a User Study". In: *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025.* The Internet Society, 2025. URL: https://www.ndss-symposium.org/ndss-paper/translating-c-to-rust-lessons-from-a-user-study/ (cit. on pp. 2, 10, 11, 151).

[Li+24a] Xuan Li, Shuai Yuan, Xiaodong Gu, Yuting Chen, and Beijun Shen. "Few-Shot Code Translation via Task-Adapted Prompt Learning". In: *J. Syst. Softw.* 212 (2024), p. 112002. DOI: 10.1016/J.JSS.2024.112002 (cit. on p. 20).

[Li+22] Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R. Lyu. "A Closer Look into Transformer-Based Code Intelligence through Code Transformation: Challenges and Opportunities". In: *CoRR* abs/2207.04285 (2022). DOI: 10.48550/ARXIV.2207.04285 (cit. on p. 33).

[Li+24b] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel C. Briand, and Michael R. Lyu. “Exploring the Effectiveness of Llms in Automated Logging Statement Generation: An Empirical Study”. In: *IEEE Transactions on Software Engineering* 50.12 (2024), pp. 3188–3207. DOI: 10.1109/TSE.2024.3475375 (cit. on pp. 33, 45, 161).

[LO04] Chin-Yew Lin and Franz Josef Och. “ORANGE: A Method for Evaluating Automatic Evaluation Metrics for Machine Translation”. In: *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics.* Geneva, Switzerland: COLING, Aug. 2004, pp. 501–507. URL: https://www.aclweb.org/anthology/C04-1072 (cit. on p. 57).

[Lin+14] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. “Microsoft COCO: Common Objects in Context”. In: *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V.* Vol. 8693. Lecture Notes in Computer Science. Springer, 2014, pp. 740–755. DOI: 10.1007/978-3-319-10602-1_48 (cit. on p. 16).

[Liu+23] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation”. In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023.* 2023. URL: http://papers.nips.cc/paper%5C_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html (cit. on p. 66).

[LLC20] Yalin Liu, Jinfeng Lin, and Jane Cleland-Huang. “Traceability Support for Multi-Lingual Software Projects”. In: *MSR ’20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020.* ACM, 2020, pp. 443–454. DOI: 10.1145/3379597.3387440 (cit. on p. 156).

[Liu+25] Yuchen Liu, Junhao Hu, Yingdi Shan, Ge Li, Yanzhen Zou, Yihong Dong, and Tao Xie. “LLMigrate: Transforming "Lazy" Large Language Models into Efficient Source Code Migrators”. In: *CoRR* abs/2503.23791 (2025). DOI: 10.48550/ARXIV.2503.23791 (cit. on p. 11).

[Liu+24] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach Dinh Le, and David Lo. “Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues”. In: *ACM Trans. Softw. Eng. Methodol.* 33.5 (2024), 116:1–116:26. DOI: 10.1145/3643674 (cit. on p. 16).

[Lu+21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation”. In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, Virtual.* 2021. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html (cit. on pp. 29, 30, 56, 57).

[Luo+22] Xianchang Luo, Yinxing Xue, Zhenchang Xing, and Jiamou Sun. "PRCBERT: Prompt Learning for Requirement Classification Using BERT-based Pretrained Language Models". In: *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, 75:1–75:13. DOI: 10.1145/3551349.3560417 (cit. on p. 16).

[Luo+24] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. "WizardCoder: Empowering Code Large Language Models with Evol-Instruct". In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=UnUwSIgK5W (cit. on p. 18).

[Ma+23] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. "The Scope of ChatGPT in Software Engineering: A Thorough Investigation". In: *CoRR* abs/2305.12138 (2023). DOI: 10.48550/ARXIV.2305.12138 (cit. on p. 16).

[MK22] Emanuele La Malfa and Marta Kwiatkowska. "The King Is Naked: On the Notion of Robustness for Natural Language Processing". In: *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 2022, pp. 11047–11057. DOI: 10.1609/AAAI.V36I10.21353 (cit. on p. 26).

[Mar14] Robert C. Martin. "Chapter 8. SRP: The Single-Responsibility Principle". In: *Agile Software Development, Principles, Patterns, and Practices*. First edition, Pearson new international edition. Harlow: Pearson, 2014, pp. 101–105. ISBN: 978-1-292-02594-0 (cit. on p. 64).

[Mas+23] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. "On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot". In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2149–2160. DOI: 10.1109/ICSE48619.2023.00181 (cit. on pp. 3–5, 30, 35, 41, 45, 46, 136, 151, 155).

[MI14] Nicholas D. Matsakis and Felix S. Klock II. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*. ACM, 2014, pp. 103–104. DOI: 10.1145/2663171.2663188 (cit. on p. 10).

[Mat+24] Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. "On Leakage of Code Generation Evaluation Datasets". In: *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*. Association for

Computational Linguistics, 2024, pp. 13215–13223. URL: https://aclanthology.org/2024.findings-emnlp.772 (cit. on pp. 17, 36).

[MW21] Aleksandra Matulewska and Anne Wagner. “Third Space of Legal Translation: Between Protean Meanings, Legal Cultures and Communication Stratification”. In: *International Journal for the Semiotics of Law - Revue internationale de Sémiotique juridique* 34.5 (Nov. 2021), pp. 1245–1260. ISSN: 0952-8059, 1572-8722. DOI: 10.1007/s11196-020-09796-5 (cit. on pp. 1, 151).

[MVC24] Nickil Maveli, Antonio Vergari, and Shay B. Cohen. “What Can Large Language Models Capture about Code Functional Equivalence?” In: *CoRR* abs/2408.11081 (2024). DOI: 10.48550/ARXIV.2408.11081 (cit. on p. 11).

[MH18] Leland McInnes and John Healy. “UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction”. In: *CoRR* abs/1802.03426 (2018). arXiv: 1802.03426 (cit. on p. 67).

[Mic+19] Paul Michel, Xian Li, Graham Neubig, and Juan Miguel Pino. “On Evaluation of Adversarial Perturbations for Sequence-to-Sequence Models”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 3103–3114. DOI: 10.18653/V1/N19-1314 (cit. on p. 32).

[Mik+13a] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. 2013. URL: http://arxiv.org/abs/1301.3781 (cit. on pp. 13, 56).

[Mik+13b] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. “Distributed Representations of Words and Phrases and Their Compositionality”. In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a Meeting Held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 2013, pp. 3111–3119. URL: https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html (cit. on p. 13).

[MYZ13] Tomás Mikolov, Wen-tau Yih, and Geoffrey Zweig. “Linguistic Regularities in Continuous Space Word Representations”. In: *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*. The Association for Computational Linguistics, 2013, pp. 746–751. URL: https://aclanthology.org/N13-1090/ (cit. on p. 13).

[Mod22] Apostolos Modas. “Robustness and Invariance Properties of Image Classifiers”. In: *CoRR* abs/2209.02408 (2022). DOI: 10.48550/ARXIV.2209.02408 (cit. on p. 26).

[MBK23] Patrick Müller, Alexander Braun, and Margret Keuper. “Classification Robustness to Common Optical Aberrations”. In: *IEEE/CVF International Conference on Computer Vision, ICCV 2023 - Workshops, Paris, France, October 2-6, 2023*. IEEE, 2023, pp. 3634–3645. DOI: 10.1109/ICCVW60793.2023.00391 (cit. on p. 26).

[Nag+09] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for c”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. ACM, 2009, pp. 245–258. DOI: 10.1145/1542476.1542504 (cit. on p. 10).

[Nag+10] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. “CETS: Compiler Enforced Temporal Safety for C”. In: *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*. ACM, 2010, pp. 31–40. DOI: 10.1145/1806651.1806657 (cit. on p. 10).

[NF15] Sebastian Nanz and Carlo A. Furia. “A Comparative Study of Programming Languages in Rosetta Code”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 2015, pp. 778–788. DOI: 10.1109/ICSE.2015.90 (cit. on p. 44).

[Nez+24] Marianna Nezhurina, Lucia Cipolina-Kun, Mehdi Cherti, and Jenia Jitsev. “Alice in Wonderland: Simple Tasks Showing Complete Reasoning Breakdown in State-of-the-Art Large Language Models”. In: *CoRR* abs/2406.02061 (2024). DOI: 10.48550/ARXIV.2406.02061 (cit. on pp. 1, 3, 4, 35, 137, 152).

[Nij+23] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis”. In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/forum?id=iaYcJKpY2B%5C_ (cit. on p. 29).

[NNP20] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. “DifFuzz: Differential Fuzzing for Side-Channel Analysis”. In: *Software Engineering 2020, Fachtagung Des GI-Fachbereichs Softwaretechnik, 24.-28. Februar 2020, Innsbruck, Austria*. Vol. P-300. LNI. Gesellschaft für Informatik e.V., 2020, pp. 125–126. DOI: 10.18420/SE2020_37 (cit. on p. 21).

[Niu+24] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. “On Evaluating the Efficiency of Source Code Generated by LLMs”. In: *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, FORGE 2024, Lisbon, Portugal, 14 April 2024*. ACM, 2024, pp. 103–107. DOI: 10.1145/3650105.3652295 (cit. on p. 19).

[Oma+22] Marwan Omar, Soohyeon Choi, Daehun Nyang, and David Mohaisen. “Robust Natural Language Processing: Recent Advances, Challenges, and Future Directions”. In: *IEEE Access* 10 (2022), pp. 86038–86056. DOI: 10.1109/ACCESS.2022.3197769 (cit. on p. 26).

[Ouy+22] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. "Training Language Models to Follow Instructions with Human Feedback". In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. 2022. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html (cit. on pp. 14, 15).

[Pap+02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "Bleu: A Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135 (cit. on pp. 29, 57).

[PG24] Brent Pappas and Paul Gazzillo. "Semantic Analysis of Macro Usage for Portability". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, 21:1–21:12. DOI: 10.1145/3597503.3623323 (cit. on pp. 11, 65).

[Par+24] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Benjamin Ferrari-Church, and Satish Chandra. "How Much Does AI Impact Development Speed? An Enterprise-Based Randomized Controlled Trial". In: *CoRR* abs/2410.12944 (2024). DOI: 10.48550/ARXIV.2410.12944 (cit. on pp. 1, 16).

[PJ15] Timo Pawelka and Elmar Jürgens. "Is This Code Written in English? A Study of the Natural Language of Comments and Identifiers in Practice". In: *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*. IEEE Computer Society, 2015, pp. 401–410. DOI: 10.1109/ICSM.2015.7332491 (cit. on p. 156).

[Pea+25] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions". In: *Commun. ACM* 68.2 (2025), pp. 96–105. DOI: 10.1145/3610721 (cit. on p. 16).

[Pen+23] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot". In: *CoRR* abs/2302.06590 (2023). DOI: 10.48550/ARXIV.2302.06590 (cit. on p. 15).

[PBY24] Ziqian Peng, Rachel Bawden, and François Yvon. "Investigating Length Issues in Document-level Machine Translation". In: *CoRR* abs/2412.17592 (2024). DOI: 10.48550/ARXIV.2412.17592 (cit. on pp. 64, 72).

[Per+23] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. "Do Users Write More Insecure Code with AI Assistants?" In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. ACM, 2023, pp. 2785–2799. DOI: 10.1145/3576915.3623157 (cit. on pp. 1, 16).

[Pet+17] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. “NEZHA: Efficient Domain-Independent Differential Testing”. In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 615–632. DOI: 10.1109/SP.2017.27 (cit. on p. 21).

[Puk94] Friedrich Pukelsheim. “The Three Sigma Rule”. In: *The American Statistician* 48.2 (May 1994), pp. 88–91. ISSN: 0003-1305, 1537-2731. DOI: 10.1080/00031305.1994.10476030 (cit. on pp. 58, 83).

[QHW25] Jochen Quante, Jesko Hecking-Harbusch, and Matthias Woehrle. “C to Rust Translation via Large Language Models”. In: *27. Workshop Software-Reengineering Und-Evolution WSRE 2025*. 2025, pp. 21–22. URL: https://fg-sre.gi.de/fileadmin/FG/SRE/wsre2025/WSRE2025_Proceedings.pdf (cit. on pp. 6, 48, 135).

[Rad+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. “Language Models Are Unsupervised Multitask Learners”. In: *OpenAI blog* 1.8 (2019), p. 9 (cit. on pp. 14, 15).

[RKK24] Akshat Ramachandran, Souvik Kundu, and Tushar Krishna. “MicroScopiQ: Accelerating Foundational Models through Outlier-Aware Microscaling Quantization”. In: *CoRR* abs/2411.05282 (2024). DOI: 10.48550/ARXIV.2411.05282 (cit. on p. 58).

[Ren+20] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. “CodeBLEU: A Method for Automatic Evaluation of Code Synthesis”. In: *CoRR* abs/2009.10297 (2020). arXiv: 2009.10297 (cit. on pp. 29, 30, 56, 57).

[Roz+23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. “Code Llama: Open Foundation Models for Code”. In: *CoRR* abs/2308.12950 (2023). DOI: 10.48550/ARXIV.2308.12950 (cit. on pp. 17, 18).

[Roz+20] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. “Unsupervised Translation of Programming Languages”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, Virtual*. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html (cit. on p. 21).

[Rus+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. “ImageNet Large Scale Visual Recognition Challenge”. In: *Int. J. Comput. Vis.* 115.3 (2015), pp. 211–252. DOI: 10.1007/S11263-015-0816-Y (cit. on pp. 3, 16).

[SWY75a] Gerard Salton, Anita Wong, and Chung-Shu Yang. “A Vector Space Model for Automatic Indexing”. In: *Commun. ACM* 18.11 (1975), pp. 613–620. DOI: 10.1145/361219.361220 (cit. on p. 13).

[SWY75b] Gerard Salton, Anita Wong, and Chung-Shu Yang. “A Vector Space Model for Automatic Indexing”. In: *Commun. ACM* 18.11 (1975), pp. 613–620. DOI: 10.1145/361219.361220 (cit. on p. 56).

[Sch+23] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. “Adaptive Test Generation Using a Large Language Model”. In: *CoRR* abs/2302.06527 (2023). DOI: 10.48550/ARXIV.2302.06527 (cit. on p. 16).

[SBS18] Patrick Schober, Christa Boer, and Lothar A. Schwarte. “Correlation Coefficients: Appropriate Use and Interpretation”. In: *Anesthesia & Analgesia* 126.5 (May 2018), pp. 1763–1768. ISSN: 0003-2999. DOI: 10.1213/ANE.0000000000002864. (Visited on 03/29/2025) (cit. on pp. 92, 108, 111, 129, 142).

[Sch+21] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. “You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 2021, pp. 1559–1575. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schuster (cit. on pp. 1, 16).

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. DOI: 10.18653/V1/P16-1162 (cit. on p. 12).

[Ser16] Kosta Serebryany. “Continuous Fuzzing with libFuzzer and AddressSanitizer”. In: *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 2016, p. 157. DOI: 10.1109/SECDEV.2016.043 (cit. on p. 22).

[Sha05] Yakov Shafranovich. “Common Format and MIME Type for Comma-Separated Values (CSV) Files”. In: *RFC* 4180 (2005), pp. 1–8. DOI: 10.17487/RFC4180 (cit. on pp. 49, 50).

[SS24] Momoko Shiraishi and Takahiro Shinagawa. “Context-Aware Code Segmentation for C-to-Rust Translation Using Large Language Models”. In: *CoRR* abs/2409.10506 (2024). DOI: 10.48550/ARXIV.2409.10506 (cit. on p. 2).

[Sid+24] Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna C. S. Santos. “The Fault in our Stars: Quality Assessment of Code Generation Benchmarks”. In: *IEEE International Conference on Source Code Analysis and Manipulation, SCAM 2024, Flagstaff, AZ, USA, October 7-8, 2024*. IEEE, 2024, pp. 201–212. DOI: 10.1109/SCAM63643.2024.00028 (cit. on pp. 5, 17, 36).

[Sie+23] Julien Siebert, Daniel Seifert, Patricia Kelbert, Michael Kläs, and Adam Trendowicz. “Badgers: Generating Data Quality Deficits with Python”. In: *CoRR* abs/2307.04468 (2023). DOI: 10.48550/ARXIV.2307.04468 (cit. on p. 26).

[Sin+23] Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. “CodeFusion: A Pre-trained Diffusion Model for Code Generation”. In: *CoRR* abs/2310.17680 (2023). DOI: 10.48550/ARXIV.2310.17680 (cit. on p. 66).

[Su+24] Jianlin Su, Murtadha H. M. Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. “RoFormer: Enhanced Transformer with Rotary Position Embedding”. In: *Neurocomputing* 568 (2024), p. 127063. DOI: 10.1016/J.NEUCOM.2023.127063 (cit. on p. 13).

[Sza+23] Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. “Code Translation with Compiler Representations”. In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/forum?id=XomEU3eNeSQ (cit. on p. 64).

[Sze+14] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. “Intriguing Properties of Neural Networks”. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014. URL: http://arxiv.org/abs/1312.6199 (cit. on p. 26).

[TFU07] Barbara G Tabachnick, Linda S Fidell, and Jodie B Ullman. *Using Multivariate Statistics*. Vol. 5. pearson Boston, MA, 2007 (cit. on pp. 58, 60).

[Tam+25] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. “Bugs in Large Language Models Generated Code: An Empirical Study”. In: *Empir. Softw. Eng.* 30.3 (2025), p. 65. DOI: 10.1007/S10664-025-10614-4 (cit. on pp. 16, 21).

[Tao+24] Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. “Unraveling the Potential of Large Language Models in Code Translation: How Far Are We?” In: *CoRR* abs/2410.09812 (2024). DOI: 10.48550/ARXIV.2410.09812 (cit. on p. 149).

[Tou+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. “LLaMA: Open and Efficient Foundation Language Models”. In: *CoRR* abs/2302.13971 (2023). DOI: 10.48550/ARXIV.2302.13971 (cit. on pp. 1, 14).

[Van+22] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. “Verifying Dynamic Trait Objects in Rust”. In: *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 2022, pp. 321–330. DOI: 10.1109/ICSE-SEIP55303.2022.9794041 (cit. on p. 23).

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (cit. on pp. 1, 13, 14).

[Ven80] John Venn. "I. On the Diagrammatic and Mechanical Representation of Propositions and Reasonings". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 10.59 (1880), pp. 1–18. DOI: 10.1080/14786448008626877 (cit. on p. 117).

[Wan+19] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.* OpenReview.net, 2019. URL: https://openreview.net/forum?id=rJ4km2R5t7 (cit. on pp. 3, 16).

[Wan+23] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. "ReCode: Robustness Evaluation of Code Generation Models". In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023.* Association for Computational Linguistics, 2023, pp. 13818–13843. DOI: 10.18653/V1/2023.ACL-LONG.773 (cit. on pp. 3–6, 26, 28, 29, 32, 35, 40, 41, 45–47, 54, 66, 135, 136, 138, 151, 155, 156, 159).

[Węg18] Stanisław Węglarczyk. "Kernel Density Estimation and Its Application". In: *ITM Web of Conferences* 23 (2018), p. 00037. ISSN: 2271-2097. DOI: 10.1051/itmconf/20182300037. (Visited on 04/04/2025) (cit. on p. 106).

[Wei+22] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. "Emergent Abilities of Large Language Models". In: *Trans. Mach. Learn. Res.* 2022 (2022). URL: https://openreview.net/forum?id=yzkSU5zdwD (cit. on pp. 1, 12, 14).

[Wil45] Frank Wilcoxon. "Individual Comparisons by Ranking Methods". In: *Biometrics Bulletin* 1.6 (Dec. 1945), p. 80. ISSN: 00994987. DOI: 10.2307/3001968. (Visited on 05/03/2025) (cit. on p. 148).

[WAV20] Winston Wu, Dustin Arendt, and Svitlana Volkova. "Evaluating Neural Machine Comprehension Model Robustness to Noisy Inputs and Adversarial Attacks". In: *CoRR* abs/2005.00190 (2020). arXiv: 2005.00190 (cit. on p. 32).

[XZ22] Chunqiu Steven Xia and Lingming Zhang. "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022.* ACM, 2022, pp. 959–971. DOI: 10.1145/3540250.3549101 (cit. on p. 16).

[YBS24] Ankit Yadav, Himanshu Beniwal, and Mayank Singh. "PythonSaga: Redefining the Benchmark to Evaluate Code Generating LLMs". In: *Findings of the Association for Computational Linguistics: EMNLP 2024.* Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 17113–17126. DOI: 10.18653/v1/2024.findings-emnlp.996 (cit. on pp. 5, 17, 36).

[Yan+23a] Ming Yan, Junjie Chen, Jie M. Zhang, Xuejie Cao, Chen Yang, and Mark Harman. "COCO: Testing Code Generation Systems via Concretized Instructions". In: *CoRR* abs/2308.13319 (2023). DOI: 10.48550/ARXIV.2308.13319 (cit. on pp. 3–5, 30, 35, 41, 45, 46, 136, 151, 156).

[Yan+23b] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. "CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation". In: *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*. Association for Computational Linguistics, 2023, pp. 5067–5089. DOI: 10.18653/V1/2023.FINDINGS-EMNLP.337 (cit. on p. 17).

[Yan+24a] Aidan Z. H. Yang, Sophia Kolak, Vincent J. Hellendoorn, Ruben Martins, and Claire Le Goues. "Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models". In: *CoRR* abs/2404.15236 (2024). DOI: 10.48550/ARXIV.2404.15236 (cit. on pp. 2, 151).

[Yan+24b] Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. "VERT: Verified Equivalent Rust Transpilation with Few-Shot Learning". In: *CoRR* abs/2404.18852 (2024). DOI: 10.48550/ARXIV.2404.18852 (cit. on pp. 2, 11, 19–21, 23, 64, 72, 140, 151).

[Yan+24c] An Yang et al. "Qwen2.5 Technical Report". In: *CoRR* abs/2412.15115 (2024). DOI: 10.48550/ARXIV.2412.15115 (cit. on p. 18).

[Yan+24d] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Ben Hu. "Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond". In: *ACM Trans. Knowl. Discov. Data* 18.6 (2024), 160:1–160:32. DOI: 10.1145/3649506 (cit. on p. 16).

[YOT22] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. "Assessing the Quality of GitHub Copilot's Code Generation". In: *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2022, Singapore, Singapore, 17 November 2022*. ACM, 2022, pp. 62–71. DOI: 10.1145/3558489.3559072 (cit. on p. 16).

[YWW22] Shiwen Yu, Ting Wang, and Ji Wang. "Data Augmentation by Program Transformation". In: *Journal of Systems and Software* 190 (Aug. 2022), p. 111304. ISSN: 01641212. DOI: 10.1016/j.jss.2022.111304. (Visited on 01/28/2025) (cit. on pp. 33, 45).

[08] "Z-Score". In: *Encyclopedia of Public Health*. Dordrecht: Springer Netherlands, 2008, pp. 1484–1484. ISBN: 978-1-4020-5614-7. DOI: 10.1007/978-1-4020-5614-7_3826 (cit. on p. 58).

[Zen+24] Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. "CoderUJB: An Executable and Unified Java Benchmark for Practical Programming Scenarios". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*. ACM, 2024, pp. 124–136. DOI: 10.1145/3650212.3652115 (cit. on p. 18).

[Zha+24a] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. "Scalable, Validated Code Translation of Entire Projects Using Large Language Models". In: *CoRR* abs/2412.08035 (2024). DOI: 10.48550/ARXIV.2412.08035 (cit. on p. 2).

[Zha+23a] Jianzhang Zhang, Yiyang Chen, Nan Niu, and Chuang Liu. "A Preliminary Evaluation of ChatGPT in Requirements Information Retrieval". In: *CoRR* abs/2304.12562 (2023). DOI: 10.48550/ARXIV.2304.12562 (cit. on p. 16).

[Zha+20] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. "PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 11328–11339. URL: http://proceedings.mlr.press/v119/zhang20ae.html (cit. on p. 30).

[Zha+23b] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. "Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations". In: *IEEE Trans. Software Eng.* 49.5 (2023), pp. 3052–3070. DOI: 10.1109/TSE.2023.3240118 (cit. on pp. 33, 45, 159).

[Zha+22] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. "Towards Understanding the Runtime Performance of Rust". In: *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, 140:1–140:6. DOI: 10.1145/3551349.3559494 (cit. on p. 10).

[Zha+24b] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. "Explainability for Large Language Models: A Survey". In: *ACM Transactions on Intelligent Systems and Technology* 15.2 (2024), 20:1–20:38. DOI: 10.1145/3639372 (cit. on p. 62).

[Zha+23c] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. "A Survey of Large Language Models". In: *CoRR* abs/2303.18223 (2023). DOI: 10.48550/ARXIV.2303.18223 (cit. on pp. 1, 9, 12–15).

[Zhe+23] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. "A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends". In: *CoRR* abs/2311.10372 (2023). DOI: 10.48550/ARXIV.2311.10372 (cit. on pp. 5, 9, 18, 19, 29, 36).

[Zho+23] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. "CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*. Association for Computational Linguistics, 2023, pp. 13921–13937. DOI: 10.18653/V1/2023.EMNLP-MAIN.859 (cit. on p. 148).

[Zho+25a] Tianyang Zhou, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. "LLM-Driven Multi-step Translation from C to Rust Using Static Analysis". In: *CoRR* abs/2503.12511 (2025). DOI: 10.48550/ARXIV.2503.12511 (cit. on p. 19).

[Zho+25b] Xin Zhou, Martin Weyssow, Ratnadira Widyasari, Ting Zhang, Junda He, Yunbo Lyu, Jianming Chang, Beiqi Zhang, Dan Huang, and David Lo. "LessLeak-Bench: A First Investigation of Data Leakage in LLMs Across 83 Software Engineering Benchmarks". In: *CoRR* abs/2502.06215 (2025). DOI: 10.48550/ARXIV.2502.06215 (cit. on p. 17).

[Zhu+22] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. "XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence". In: *CoRR* abs/2206.08474 (2022). DOI: 10.48550/ARXIV.2206.08474 (cit. on p. 17).

[Zie+24] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. "Measuring GitHub Copilot's Impact on Productivity". In: *Commun. ACM* 67.3 (2024), pp. 54–63. DOI: 10.1145/3633453 (cit. on p. 16).

Web Pages and Software

[Abd+25] Marah I Abdin, Jyoti Aneja, Harkirat S. Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. *Microsoft/Phi-4-Gguf · Hugging Face*. Accessed on 2025-05-11. May 2025. URL: https://huggingface.co/microsoft/phi-4-gguf (cit. on p. 66).

[AUT] AUTOSAR. *Automotive Open System Architecture, Classic Platform*. Accessed on 2025-03-30. URL: https://www.autosar.org/standards/classic-platform (cit. on p. 64).

[Bru+25] Max Brunsfeld et al. *Tree-Sitter/Tree-Sitter: V0.25.2*. Feb. 2025. DOI: 10.5281/ZENODO.14885565 (cit. on pp. 149, 153, 157).

[Cas24] Stephen Cass. *Top Programming Languages 2024 - IEEE Spectrum*. Accessed on 2025-04-15. 2024. URL: https://spectrum.ieee.org/top-programming-languages-2024 (cit. on p. 35).

[Cim19] Catalin Cimpanu. *Microsoft: 70 Percent of All Security Bugs Are Memory Safety Issues*. Accessed on 2025-04-14. 2019. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ (cit. on pp. 2, 10).

[Cod24] Codacy. *10 Common Programming Errors and How to Avoid Them*. Accessed on 2025-04-03. 2024. URL: https://blog.codacy.com/common-programming-errors (cit. on p. 104).

[Cod25a] Papers with Code. *GLUE Dataset*. Accessed on 2025-04-20. 2025. URL: https://paperswithcode.com/dataset/glue (cit. on p. 16).

[Cod25b] Papers with Code. *HumanEval Benchmark (Code Generation)*. Accessed on 2025-05-03. 2025. URL: https://paperswithcode.com/sota/code-generation-on-humaneval (cit. on p. 147).

[Cod25c] Papers with Code. *ImageNet Benchmark (Image Classification)*. Accessed on 2025-04-20. 2025. URL: https://paperswithcode.com/sota/image-classification-on-imagenet (cit. on p. 16).

[Cod25d] Papers with Code. *MBPP Benchmark (Code Generation)*. Accessed on 2025-04-20. 2025. URL: https://paperswithcode.com/sota/code-generation-on-mbpp (cit. on p. 16).

[Cod25e] Papers with Code. *MS COCO Dataset*. Accessed on 2025-04-20. 2025. URL: https://paperswithcode.com/dataset/coco (cit. on p. 16).

[Cod] Codeforces. *Programming competitions and contests*. Accessed on 2025-05-11. URL: https://codeforces.com/ (visited on 05/11/2025) (cit. on p. 17).

[DAR24] DARPA. *TRACTOR: Translating All C to Rust*. Accessed on 2025-04-14. 2024. URL: https://www.darpa.mil/research/programs/translating-all-c-to-rust (cit. on pp. 2, 9, 10, 151).

[Far17] Aleks Farseev. *Is Bigger Better? Why The ChatGPT Vs. GPT-3 Vs. GPT-4 'Battle' Is Just A Family Chat*. Accessed on 2025-04-08. 2017. URL: https://www.forbes.com/councils/forbestechcouncil/2023/02/17/is-bigger-better-why-the-chatgpt-vs-gpt-3-vs-gpt-4-battle-is-just-a-family-chat/ (cit. on p. 66).

[Fow15] Martin Fowler. *Bliki: Yagni*. Accessed on 2025-05-11. 2015. URL: https://martinfowler.com/bliki/Yagni.html (cit. on p. 48).

[Gee24] GeeksforGeeks. *Add Two Numbers without Using Arithmetic Operators*. Accessed on 2025-04-03. 2024. URL: https://www.geeksforgeeks.org/add-two-numbers-without-using-arithmetic-operators/ (cit. on p. 56).

[GNUa] GNU. *Names (GNU Coding Standards)*. Accessed on 2025-05-10. URL: https://www.gnu.org/prep/standards/html_node/Names.html (cit. on p. 158).

[GNUb] GNU. *Unions (GNU C Language Manual)*. Accessed on 2025-05-11. URL: https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Unions.html (cit. on p. 11).

[Gro21] Internet Security Research Group. *What Is Memory Safety and Why Does It Matter?* Accessed on 2025-04-16. 2021. URL: https://www.memorysafety.org/docs/memory-safety/ (cit. on p. 10).

[Hui+25] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. *Qwen/Qwen2.5-Coder-14B-Instruct-GGUF · Hugging Face*. Accessed on 2025-05-11. Apr. 2025. URL: https://huggingface.co/Qwen/Qwen2.5-Coder-14B-Instruct-GGUF (cit. on p. 66).

[IBM25] IBM. *Standard C Library Functions Table, By Name*. Accessed on 2025-05-11. Apr. 2025. URL: https://www.ibm.com/docs/en/i/7.5.0?topic=extensions-standard-c-library-functions-table-by-name (cit. on p. 160).

[Imm19a] Immunant. *C2Rust Manual*. Accessed on 2025-04-15. 2019. URL: https://c2rust.com/manual/ (cit. on p. 2).

[Imm19b] Immunant. *C2Rust: Migrate C Code to Rust*. Accessed on 2025-04-14. 2019. URL: https://github.com/immunant/c2rust (cit. on p. 11).

[Kat] Kattis. *Problem Archive*. Accessed on 2025-05-11. URL: https://open.kattis.com/ (visited on 05/11/2025) (cit. on p. 17).

[Keh19] Paul Kehrer. *Memory Unsafety in Apple's Operating Systems*. Accessed on 2025-04-16. 2019. URL: https://langui.sh/2019/07/23/apple-memory-safety/ (cit. on p. 10).

[Lan] The Rust Programming Language. *Introduction - Clippy Documentation*. Accessed on 2025-05-11. URL: https://doc.rust-lang.org/clippy/ (cit. on p. 51).

[LLVa] LLVM. *Clang - Expressive Diagnostics*. Accessed on 2025-05-11. URL: https://clang.llvm.org/diagnostics.html (cit. on p. 155).

[LLVb] LLVM. *ClangFormat — Clang 21.0.0 git Documentation*. Accessed on 2025-05-10. URL: https://clang.llvm.org/docs/ClangFormat.html (cit. on p. 157).

[LLVc] LLVM. *Coding Standards — LLVM 21.0.0git Documentation*. Accessed on 2025-05-10. URL: https://llvm.org/docs/CodingStandards.html (cit. on pp. 157, 158).

[Mic25] Microsoft. *What Is Azure OpenAI Service?* Accessed on 2025-05-11. 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/openai/overview (cit. on p. 65).

[Mic] Microsoft. *Azure OpenAI Service – Pricing*. Accessed on 2025-05-10. URL: https://azure.microsoft.com/de-de/pricing/details/cognitive-services/openai-service/ (cit. on p. 65).

[Moza] Mozilla. *400 Bad Request*. Accessed on 2025-05-12. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/400 (cit. on p. 74).

[Mozb] Mozilla. *500 Internal Server Error*. Accessed on 2025-05-12. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/500 (cit. on p. 74).

[Mozc] Mozilla. *Coding Style — Firefox Source Docs Documentation*. Accessed on 2025-05-10. URL: https://firefox-source-docs.mozilla.org/code-quality/coding-style/index.html (cit. on p. 157).

[Ope22a] OpenAI. *ChatGPT*. Accessed on 2025-04-14. 2022. URL: https://chat.openai.com (cit. on p. 1).

[Ope22b] OpenAI. *Introducing ChatGPT*. Accessed on 2025-04-19. 2022. URL: https://openai.com/index/chatgpt/ (cit. on p. 14).

[Ope22c] OpenAI. *New and improved embedding model*. Accessed on 2025-04-03. 2022. URL: https://openai.com/index/new-and-improved-embedding-model/ (cit. on pp. 57, 105, 113, 141).

[Ope23] OpenAI. *GPT-3.5 Turbo Fine-Tuning and API Updates*. Accessed on 2025-03-30. Aug. 2023. URL: https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/ (cit. on pp. 7, 18, 65).

[Ope24a] OpenAI. *GPT-4o Mini: Advancing Cost-Efficient Intelligence*. Accessed on 2025-03-30. July 2024. URL: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ (cit. on pp. 6, 18, 65, 157).

[Ope24b] OpenAI. *GPT-4o System Card*. Accessed on 2025-03-30. 2024. URL: https://openai.com/index/gpt-4o-system-card/ (cit. on p. 65).

[Ope24c] OpenAI. *Introducing OpenAI O1*. Accessed on 2025-03-30. Sept. 2024. URL: https://openai.com/o1/ (cit. on p. 65).

[Ope25a] OpenAI. *Introducing 4o Image Generation*. Accessed on 2025-05-11. 2025. URL: https://openai.com/index/introducing-4o-image-generation/ (cit. on p. 168).

[Ope25b] OpenAI. *Introducing OpenAI O3 and O4-Mini*. Accessed on 2025-05-03. 2025. URL: https://openai.com/index/introducing-o3-and-o4-mini/ (cit. on p. 149).

[Ope25c] OpenAI. *OpenAI/Tiktoken*. OpenAI. Accessed on 2025-05-12. May 2025. URL: https://github.com/openai/tiktoken (cit. on pp. 13, 64).

[Ope] OpenAI. *Vector Embeddings - OpenAI API*. Accessed on 2025-05-11. URL: https://platform.openai.com/docs/guides/embeddings (cit. on p. 67).

[Oraa] Oracle. *ArrayIndexOutOfBoundsException (Java Platform SE 7)*. Accessed on 2025-05-11. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/ArrayIndexOutOfBoundsException.html (cit. on p. 10).

[Orab] Oracle. *Javadoc*. Accessed on 2025-05-14. URL: https://www.oracle.com/java/technologies/javase/javadoc.html (cit. on p. 30).

[Pat25] PatrickFarley. *Prompt Shields in Azure AI Content Safety - Azure AI Services*. Accessed on 2025-05-11. 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/content-safety/concepts/jailbreak-detection (cit. on p. 87).

[San20] Silvia Sanchez. *C Is for Car - Safety Critical Systems in the Automotive Industry*. Accessed on 2025-04-14. 2020. URL: https://www.qa-systems.com/blog/c-is-for-car/ (cit. on p. 2).

[Sta23] Inbal Shani Staff GitHub. *Survey Reveals AI's Impact on the Developer Experience*. Accessed on 2025-01-07. June 2023. URL: https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/ (visited on 01/07/2025) (cit. on pp. 1, 16).

[The21] The Rust Programming Language. *To Panic! Or Not to Panic!* Accessed on 2025-05-05. 2021. URL: https://doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html (cit. on p. 10).

[Thea] The Rust Programming Language. *Naming - Rust API Guidelines*. Accessed on 2025-05-10. URL: https://rust-lang.github.io/api-guidelines/naming.html (cit. on p. 158).

[Theb] The Rust Programming Language. *Unsafe Rust*. Accessed on 2025-05-08. URL: https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html (cit. on p. 150).

[Thec] The Rust Programming Language. *What Is Rustc? - The Rustc Book*. Accessed on 2025-05-11. URL: https://doc.rust-lang.org/rustc/what-is-rustc.html (cit. on p. 51).

[Tom20] Federico Tomassetti. *How to Write a Transpiler*. Accessed on 2025-04-15. 2020. URL: https://tomassetti.me/how-to-write-a-transpiler/ (cit. on p. 11).

[Tou+24] Hugo Touvron et al. *Meta-Llama/Llama-2-7b · Hugging Face*. Accessed on 2025-05-10. Dec. 2024. URL: https://huggingface.co/meta-llama/Llama-2-7b (cit. on p. 158).

[Wan21] Ben Wang. *Mesh-Transformer-JAX: Model-Parallel Implementation of Transformer Language Model with JAX*. Accessed on 2025-01-30. May 2021. URL: https://github.com/kingoflolz/mesh-transformer-jax (cit. on p. 29).

[Web] WebAssembly. *WebAssembly (abbreviated Wasm)*. Accessed on 2025-05-12. URL: https://webassembly.org/ (visited on 05/12/2025) (cit. on p. 23).

[Wik25a] Wikipedia. *De Morgan's Laws*. Accessed on 2025-05-10. May 2025. URL: https://en.wikipedia.org/w/index.php?title=De_Morgan%27s_laws&oldid=1289573286 (cit. on p. 161).

[Wik25b] Wikipedia. *QWERTY*. Accessed on 2025-05-12. Apr. 2025. URL: https://en.wikipedia.org/w/index.php?title=QWERTY (cit. on p. 156).

[Wik25c] Wikipedia. *Short-Circuit Evaluation.* Accessed on 2025-05-10. Apr. 2025. URL: https://en.wikipedia.org/w/index.php?title=Short-circuit_evaluation&oldid=1286047126 (cit. on p. 161).

[Zha19] Jeff Vander Stoep Chong Zhang. *Queue the Hardening Enhancements.* Accessed on 2025-04-16. 2019. URL: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (cit. on p. 10).

[µOS14] µOS++. *C/C++ Naming Conventions.* Accessed on 2025-05-10. Feb. 2014. URL: https://micro-os-plus.github.io/develop/naming-conventions/ (cit. on p. 158).

Appendix: STATEMENT OF AUTHORSHIP of the Student

Thesis:

Name:	Surname:
Date of birth:	Matriculation no.:

I herewith assure that I have written the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or according to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Furthermore, I confirm that I am aware that the use of content (including but not limited to text, figures, images and code) generated by artificial intelligence (AI) in the thesis must be disclosed. In those cases I have specified the AI system used, I have marked the specific sections of the thesis where AI-generated content was used, and I have provided a brief explanation of the level of detail at which the AI system was used to generate the content. I also stated the reason for using the tools.

I assure that even if a generative AI system has been used, the scientific contribution has been made entirely by myself.

Magdeburg, 15.05.2025

____
Signature