FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Framework for Evaluating the Correctness of Programming Learning Platforms Feedback

André Miguel Teixeira Pestana

Mestrado em Engenharia de Software

Supervisor: Prof. Ana C. R. Paiva, PhD

Second Supervisor: Prof. Auri Vincenzi, PhD

July 23, 2026

A Framework for Evaluating the Correctness of Programming Learning Platforms Feedback

André Miguel Teixeira Pestana

Mestrado em Engenharia de Software

Approved in oral examination by the committee:

President: Prof. Jorge Melegati
Referee: Prof. Nuno Macedo
Referee: Prof. Ana Paiva

July 23, 2026

Resumo

As plataformas de aprendizagem de programação online são amplamente utilizadas para apoiar a aquisição de competências de programação, fornecendo feedback automatizado sobre as soluções submetidas. Apesar da sua popularidade, a correção deste feedback é normalmente assumida, em vez de verificada de forma sistemática. Isto levanta uma questão importante: até que ponto estas plataformas podem ser consideradas fiáveis na avaliação da correção de programas?

Esta dissertação investiga se os Modelos de Linguagem de Grande Escala (LLMs) podem servir como ferramentas de validação independentes para avaliar a fiabilidade do feedback fornecido por plataformas de aprendizagem de programação. Em particular, explora se especificações de problemas em linguagem natural, quando combinadas com geração de testes baseada em LLMs e técnicas de raciocínio formal, podem revelar inconsistências nas avaliações das plataformas.

Para tal, é proposto um fluxo de trabalho em cinco etapas: (1) recolha de desafios de programação, especificações de problemas e implementações de referência a partir de uma plataforma online; (2) geração de casos de teste baseados na especificação utilizando LLMs; (3) derivação de pré-condições e pós-condições para formalizar as especificações; (4) avaliação da generalização da estratégia de prompting entre diferentes categorias de problemas; e (5) refinamento iterativo dos prompts para melhorar a qualidade da geração de testes.

A avaliação empírica abrange 219 desafios de programação, incluindo 207 da categoria Iniciante e 12 de outras categorias. Os resultados mostram que, embora a maioria das soluções aceites esteja correta, a abordagem proposta identifica vários casos em que implementações incorretas foram aceites pela plataforma. Além disso, provas formais geradas por LLM confirmam a maioria das inconsistências detetadas através da geração de testes, reforçando a fiabilidade dos resultados.

O estudo também demonstra que a estrutura dos prompts influencia significativamente a qualidade dos casos de teste gerados. Após o refinamento dos prompts, a Taxa de Sucesso de Execução de Testes aumentou de 54,86% para 70,9%, indicando uma melhoria na robustez dos conjuntos de testes gerados.

De forma geral, os resultados sugerem que os LLMs podem servir como ferramentas auxiliares eficazes na avaliação da correção do feedback automatizado em plataformas de aprendizagem de programação, ao mesmo tempo que evidenciam a importância do desenho de prompts e da qualidade das especificações em fluxos de trabalho de testes baseados em LLMs

Abstract

Online programming learning platforms are widely used to support the acquisition of programming skills by providing automated feedback on submitted solutions. Despite their popularity, the correctness of this feedback is typically assumed rather than systematically verified. This raises an important question: to what extent can such platforms be trusted to correctly assess program correctness?

This dissertation investigates whether Large Language Models (LLMs) can serve as independent validation tools to assess the reliability of feedback from programming learning platforms. In particular, it explores whether natural language problem specifications, when combined with LLM-based test generation and formal reasoning techniques, can reveal inconsistencies in platform evaluations.

To address this, a five-step workflow is proposed: (1) collection of programming challenges, problem specifications, and reference implementations from an online platform; (2) generation of specification-based test cases using LLMs; (3) derivation of preconditions and postconditions to formalize specifications; (4) evaluation of the generalizability of the prompting strategy across different problem categories; and (5) iterative refinement of prompts to improve test generation quality.

The empirical evaluation covers 219 programming challenges, including 207 from a Beginner category and 12 from additional categories. The results show that, while most accepted solutions are correct, the proposed approach identifies multiple cases where incorrect implementations were accepted by the platform. Furthermore, LLM-generated formal proofs confirm most of the inconsistencies detected through test generation, strengthening the reliability of the findings.

The study also demonstrates that prompt structure significantly influences the quality of generated test cases. After prompt refinement, the Test Execution Success Rate increased from 54.86% to 70.9%, indicating improved robustness of the generated test suites.

Overall, the results suggest that LLMs can serve as effective auxiliary tools for evaluating the correctness of automated feedback in programming learning platforms, while also highlighting the importance of prompt design and specification quality in LLM-based testing workflows.

Acknowledgements

Firstly, I would like to thank my supervisors. Their guidance and support were invaluable throughout this journey, and it was a privilege to work alongside such brilliant minds.

I would like to thank my grandparents. Every achievement, both academic and personal, would not have been possible without them. I owe them everything, and I could not have asked for better people to guide me through life.

Laura, thank you for being there through the ups and downs. Your presence, our long conversations, and the life and novelty you brought into my everyday routine meant more to me than I can express. May we always have a bit of Salsa and the stars above us.

I would also like to thank all my friends who have supported me, directly or indirectly, throughout this master's degree. Thank you, Vítor, Leo, Zé, Pedro, and Miguel. It is a genuine pleasure to call you my friends. You are all very special to me, and despite the Atlantic Ocean between some of us, know that you can count on me for anything.

Martial arts have been a part of my life since I was eight years old. What started as a hobby quickly became a passion, although it was briefly interrupted. I would like to thank Sensei Rafa for teaching me everything I know about Karate. It was more than ten years filled with joy, victories and defeats, and, more importantly, life lessons. I would also like to thank Mestre Gregory for reigniting a flame that was once extinguished. Jiu-Jitsu entered my life during a difficult period, and I believe that without it I would not have been able to overcome what seemed like an unbreakable wall. Jiu-Jitsu taught me that pressure is not something to fear, but something to understand. It gave me the ability to face uncertainty and life's challenges with resilience rather than fear.

Finally, I acknowledge that I had the opportunity to continue my studies when others have not. I consider myself fortunate in this regard, and I do not take anything I have for granted.

I dedicate this final academic work to the person who most supported me in my life endeavours.

Avô, isto é para ti.

André Miguel Teixeira Pestana

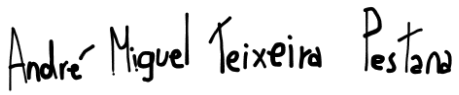
"Pressure is a privilege"

Roger Gracie

Declaração de Honra

Declaro que a presente dissertação é de minha autoria e não foi previamente apresentada e avaliada nesta ou em outra instituição. As referências a outros autores (afirmações, ideias, pensamentos), assim como a utilização de trabalhos publicados respeitam escrupulosamente as regras da atribuição, e encontram-se devidamente indicadas no texto e nas referências bibliográficas, de acordo corn as normas de referenciação. Tenho consciência de que a prática de plágio ou de autoplágio constituem ilícitos académicos, conforme estabelecido no Código de Ética da U.Porto.
Declaro, ainda, que utilizei ferramentas de Inteligência Artificial para a realização de parte(s) da presente dissertação, e essa utilização e os seus resultados estão devidamente assinalados e foram objeto de cuidada verificação para deteção de erros, imprecisões, atribuições infundadas ou outras formas de enviesamento de conteúdos e argumentos, bem como de determinação de autoria.

André Miguel Teixeira Pestana



Contents

List of Figures

List of Tables

List of Acronyms

LLM	Large Language Model
TESR	Test Execution Success Rate

Chapter 1

Introduction

This chapter presents the motivation and background of this dissertation, as well as the relevance of the proposed approach. It also outlines the research questions guiding the study and the overall contributions of this research.

1.1 Study Motivation and Background

In software testing, the importance of thorough and efficient testing practices is widely recognised. Consequently, the central challenge is no longer whether software should be tested, but rather how testing activities can be performed as effectively, reliably, and efficiently as possible.

In recent years, Large Language Models (LLMs) have emerged as promising tools for supporting both code and test generation, enabling greater automation and accelerating software development. Despite these advantages, concerns regarding the reliability and quality of the generated test suites remain significant [20]. Long before the emergence of AI-assisted testing, researchers and practitioners were already confronted with the Oracle Problem [1], namely the challenge of determining whether a test suite can correctly assess the behaviour of a software system. This challenge has not disappeared with the introduction of AI; rather, it has evolved into a new form.

Instead of being restricted to developer-written specifications, assertions, or manually designed test cases, the Oracle Problem now extends to AI-generated testing artefacts, raising important questions regarding the correctness, completeness, and trustworthiness of LLM-generated outputs. Previous studies have shown that test cases generated solely from source code often lack contextual relevance and accuracy [20]. Furthermore, even approaches that leverage detailed natural language specifications may still produce test cases that fail to accurately capture the intended behaviour of the system under test or provide adequate behavioural coverage [17]. This persistent gap between the intended specification and the generated tests highlights the need to revisit the Oracle Problem within the context of AI-driven software testing.

The above challenge is evident in several domains, some more explored than others; there is a lack of discussion about how this can affect the online learning process. Nowadays, learning has shifted slightly, and there are platforms that provide unsupervised learning, where students

submit their solutions to the exercises presented, and the feedback for those solutions is provided solely by the platform. The question at hand is how we can ensure that online platforms' oracles are correct. It was identified in previous research on selecting the best online course [26] and on assessing the usefulness and effectiveness of online platforms [13, 19], but in this research, it was not determined whether the feedback provided by these teaching platforms is correct.

This dissertation seeks to further investigate whether we can trust the feedback given by online platforms and, along with this analysis, to propose a framework that will assist in evaluating this question. More specifically, the study evaluates the synergy among specification-based test generation, formal specification derivation, formal verification, and prompt structuring in the context of an online educational programming platform comprising nine categories and 2,411 challenges.

The empirical evaluation focuses on 219 challenges. Of these, 207 belong to the "Beginner" category, while an additional 12 challenges were selected from four distinct categories: "Ad-Hoc", "Mathematics", "Data Structures and Libraries", and "Strings". Furthermore, inspired by self-repair mechanisms previously explored in the literature [6], this dissertation investigates whether obstacles identified during oracle generation can be leveraged to iteratively refine the prompting strategy and improve Test Execution Success Rate. In doing so, the study not only evaluates the effectiveness of problem specifications as test oracles but also examines the extent to which iterative improvement mechanisms can enhance the reliability of AI-assisted test-generation workflows.

1.2 Relevance of the Study

The relevance of this study stems from the growing role of online programming learning platforms in modern education. Thousands of learners rely on these platforms to develop programming skills and, in many cases, treat the feedback provided by their automated evaluation systems as an authoritative indication of correctness. Despite their widespread adoption, the reliability of such feedback is rarely questioned or systematically assessed. Consequently, understanding whether accepted solutions are indeed correct constitutes an important challenge for both programming education and software quality assurance.

This dissertation addresses this challenge by investigating whether Large Language Models can assist in the independent evaluation of platform feedback. More specifically, it explores whether specification-based test generation and formal verification techniques can identify inconsistencies in solutions accepted by an online programming platform. In doing so, the study contributes to a better understanding of the extent to which automated educational assessment systems can be trusted and how additional validation mechanisms may strengthen confidence in their results.

From a research perspective, this work is also relevant because it examines the combined use of several AI-assisted verification techniques that are typically studied independently. By integrating specification-based test generation, formal specification derivation, formal proofs, and prompt refinement into a unified workflow, the dissertation provides empirical evidence regarding the effectiveness, limitations, and practical applicability of these approaches when used together.

Finally, the study contributes to the broader discussion surrounding the use of Generative Artificial Intelligence in software testing and verification. Beyond evaluating the reliability of educational platforms, the findings offer insights into how LLMs can support software validation tasks, identify defects, and iteratively improve their own outputs through prompt refinement. As such, this research contributes to both software engineering and computing education while addressing an important gap in the current literature.

1.3 Research Questions

To address the identified research gap, this dissertation investigates the following research questions:

- **RQ1: Can we trust the feedback provided by programming learning platforms?**

  This research question aims to investigate whether programming learning platforms exhibit inconsistencies in evaluating submitted solutions. More specifically, it seeks to determine whether such platforms may incorrectly classify defective solutions as correct.

- **RQ2: Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of test cases?**

  This research question extends RQ1 by investigating whether test cases generated by Large Language Models can be used to identify incorrect solutions accepted by programming learning platforms. It evaluates the effectiveness of specification-based test generation as a mechanism for detecting defects in accepted implementations.

- **RQ3: Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of formal proofs?**

  This research question further extends RQ1 by exploring whether LLM-generated formal specifications and formal proofs can be used to assess the correctness of accepted solutions. In particular, it investigates whether formal verification artefacts generated from problem specifications can identify inconsistencies or defects that may not be immediately apparent through conventional testing alone.

- **RQ4: Could the prompt structure affect the results regarding the test cases generated by the LLM?**

  This research question builds upon RQ2 and aims to evaluate the impact of prompt structure on the effectiveness of the generated test suites. Specifically, it investigates whether prompt refinement can improve test case quality, increase the Test Execution Success Rate, and enhance the ability of generated tests to identify defects.

1.4 Main Contributions

This section summarises the main contributions of this dissertation, which emerge from the proposed methodology and its experimental evaluation. These contributions address the identified research gap and are grounded in the results obtained across all phases of the study.

The main contributions can be summarized as follows:

- A largely autonomous workflow requiring minimal human intervention and therefore supporting scalability across a large number of programming challenges.
- A unified approach capable of identifying incorrect solutions and improving the robustness of initially generated test suites through the combination of specification-based testing and formal verification techniques.
- Empirical evidence demonstrating that prompt structure significantly influences the quality and reliability of LLM-generated test cases, affecting both execution success rates and defect detection capability.

1.5 Dissertation Structure

In addition to this introductory chapter, the dissertation is structured into four further chapters.

Chapter 2 presents the literature review, examining existing research related to the topics addressed in this dissertation. It also identifies the research gap motivating this work and highlights areas requiring further investigation to advance the current body of knowledge.

Chapter 3 introduces the online platform used throughout the study and provides a detailed description of the proposed framework, including all phases of the methodology and their respective objectives.

Chapter 4 presents the experimental study, offering a comprehensive account of the procedures performed in each phase, the artefacts generated, and the data collected throughout the experimentation process.

Finally, Chapter 5 discusses the main findings of the dissertation, revisits the research questions in light of the obtained results, and explains how the proposed work contributes to addressing the identified research gap. The chapter concludes by outlining the study's limitations and presenting opportunities for future research in this area.

1.6 Disclaimer

In this dissertation, large language models were used to assist with language refinement, grammar correction, and the improvement of textual clarity. All scientific content, literature analysis, proposed approach, experiment results, and conclusions were produced and validated by the author.

All figures presented in Chapter 4 were generated with the assistance of an LLM. The original data for each challenge, including the number of test cases, Test Execution Rate percentage, proof

outcomes, difficulty levels, challenge IDs, and all analysed categories, were manually collected and stored in Google Sheets. This data was subsequently provided to the LLM to generate visualisations. Each generated figure was carefully reviewed to ensure it accurately and truthfully represented the underlying data.

Chapter 2

Related Work

This chapter introduces the relevant literature for this dissertation. It presents the literature review strategy adopted, analyses the selected studies and their main contributions, and concludes by identifying the research gap that motivates the proposed work.

2.1 Literature Review Strategy

To conduct the literature review, a systematic, structured approach was adopted to identify relevant sources while minimising the inclusion of unrelated material. To gather the body of knowledge related to this topic, Scopus was used as the data source, and the review followed the systematic approach described in [17].

2.1.1 Article sources

This literature review analyses articles on the generation of oracles using LLMs, as well as on tailored tools and frameworks for unit testing. The dataset used to gather this knowledge was Scopus, which allows for a personalised query to fetch articles specific to this topic. The chosen query is as follows:

```
1 ("test oracle" OR "oracle generation" OR "test case generation" OR "
     assertion generation" ) AND
2 ( "software testing" OR "automated testing" OR "unit testing" OR "
     context-aware testing" OR "test automation" OR "software
     verification" OR "fault detection" OR "code generation" OR "
     specification inference" OR "documentation" ) AND
3 ( "LLM" OR "large language model*" OR "language model*" OR "deep
     learning" OR "neural network*" OR "machine learning" OR "AI" OR "
     artificial intelligence" OR "natural language processing" OR "
     text-to-code")
```

This query generated 310 candidate articles as its output; this number will narrow down with further filtering undergone in the next steps.

A systematic literature selection process was applied to ensure that only relevant studies were included in the review. The process involved query-based retrieval, followed by two filtering stages and a final snowballing step to capture additional relevant work.

Table 2.1 summarises the overall selection pipeline and the resulting number of studies at each stage.

Table 2.1: Literature Selection Process

Stage	**Description**	**Number of Papers**
Initial Search	Candidate papers retrieved from the Scopus query	310
Inclusion Criteria Applied	Search query related only to keywords Publication period between 2015–2026 Conference papers or journal articles Written in English	262
Exclusion Criteria Applied	Title not related to the research area Abstract revealed a mismatch with the research topic	14
Reference Snowballing	Additional relevant studies identified through references	7
Final Literature Set	Total number of articles included in the literature review	21

2.2 Literature Review

In this section, the final selection of articles will be analysed and categorised by their contributions and methodologies, identifying patterns and challenges within each school of thought. The oracle generation problem has shifted into the AI domain, and now testers will use different artefacts, such as requirements or bug reports, and even strategies, such as combining white- and black-box testing techniques or leveraging only code with tailored prompting, to overcome this ongoing challenge. The approaches studied are summarised below. In Figure 2.1 it is identified the main schools of thought identified within the Literature.

2.2.1 Documentation-Based Oracle Generation

Documentation-based oracle generation is an approach that uses artefacts from the software under test (SUT), such as Javadoc comments, method descriptions, and developer annotations, to infer expected program behaviour and derive test oracles. The core motivation behind this line of research is that documentation often captures semantic intent that is either implicit or difficult to

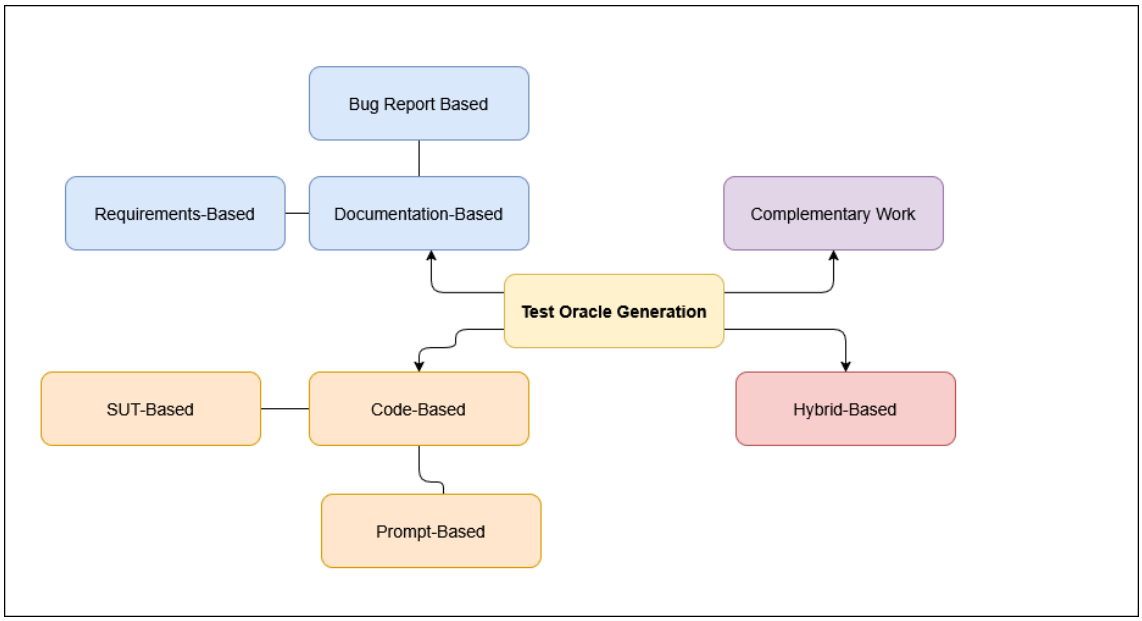


Figure 2.1: Mindmap with main schools of thought identified in the Literature

recover from source code alone. By combining these textual artefacts with contextual code information, researchers aim to mitigate the oracle problem while minimising additional effort from testers.

2.2.1.1 Requirements-Based

Requirements, written either formally or informally and in natural language, can be leveraged alongside LLMs to generate the testing oracle. Requirements deliver relevant information regarding the problem at hand and were used in recent studies [9, 17, 18] to understand to what extent this could be used and what the inherent limitations are.

Recent studies have investigated the feasibility and limitations of this approach by evaluating the requirements across different complexity levels using multiple coverage metrics [17]. The evaluation indicates that both the complexity of the requirements, and the selected coverage metric significantly influence the effectiveness of the generated test suites. In particular, requirements that are lengthy and structurally complex, quantified using Cyclomatic Complexity, tend to generate test suites that fail to adequately satisfy stricter coverage criteria. This limitation is particularly evident in Condition Coverage and Modified Condition/Decision Coverage (MCDC), where LLM-generated tests frequently achieve low coverage.

This particular study did not consider the defect-detection capability of such an approach, meaning that, despite the notable obstacle, these results can still be considered optimistic, as they were not evaluated in a more robust manner.

In contrast, another study presents a more positive result, by demonstrating that DeepSeek-Chat can effectively leverage requirements to generate meaningful test suites [18]. In this work,

requirements were used as the sole source of information to evaluate whether an LLM could produce test cases that outperform those written by developers without automated assistance. The evaluation, conducted on two real-world software systems, showed that the test suites generated using DeepSeek-Chat achieved higher defect detection rates and greater code coverage. Additionally, the authors report that the generated test cases could be seamlessly integrated into existing automated testing pipelines, enhancing practical applicability. Despite these promising results, the authors acknowledge that the experimental sample was limited, and therefore, the findings should be interpreted with caution.

Another study investigated whether natural-language specifications could be systematically translated into formal-method post-conditions using large language models [9]. The authors evaluated the correctness of the generated post-conditions, showing that they were usually consistent with the intended program behaviour and capable of distinguishing buggy versions. This was further validated on a known benchmark, Defect4J, where the generated post-conditions successfully detected a high number of real-world bugs. This work demonstrates that natural language can be leveraged to produce formal method post-conditions [9]. However, the evaluation primarily relies on test-based approximations of correctness, indicating that further investigation is required to assess robustness to incomplete or ambiguous specifications, which may negatively affect post-condition quality and defect-detection effectiveness. This line of research demonstrates that post-conditions could be applied as oracles.

In conclusion, using requirements as the sole input for oracle generation is a complex, but promising task. While some studies highlight notable improvements in coverage and defect detection, others reveal significant limitations due to requirement complexity and length, and, lastly, some research indicates that natural language specifications can be used for post-condition generation. To conclude, additional research should be done to further explore this topic and understand how different types of artefacts, such as the method under test code, could influence the defect detection rate.

2.2.1.2 Bug Report Based

This methodology can be applied to the oracle generation problem by leveraging information from user-written bug reports [15]. Such reports are written in informal natural language and often describe non-crashing, GUI-based failures, such as visual inconsistencies or incorrect interface behaviour.

Recent studies indicate that LLMs, to some extent, can use these bug descriptions and reason about the corresponding GUI structures to generate assertion-based test oracles. An example is ANDROB2O [15], which relies on the full bug report as input and combines prompt chaining, zero-shot, and chain-of-thought prompting. By breaking the oracle generation process into multiple reasoning steps, the approach aims to improve results by incorporating as much context as possible. The applicability of ANDROB2O in realistic scenarios is evaluated using bug reports unseen by the training dataset. The results indicate that the approach has some degree of generalisation, producing oracles beyond the evaluation dataset and meaningful oracles for a significant

portion of the studied failures. Moreover, the authors introduce a benchmark that can serve as a reference point for future research on oracle generation for GUI-based failures. Further research is needed to explore how different artefacts, including the method under test code, may influence defect detection rates.

Table 2.2: Documentation-Based Oracle Generation Approaches

Tool/Strategy	**Strengths**	**Limitations and Research Opportunities**
Requirements-Based	Leverages requirements written in natural language to generate test oracles. Can improve code coverage and, in some cases, defect detection. Useful for understanding intended software behaviour without requiring source code. Can be used to generate postconditions from natural-language specifications.	Performance decreases for lengthy or structurally complex requirements. Lower effectiveness in stricter coverage criteria, such as Condition Coverage and MCDC. Requires stronger evaluation of defect detection capabilities. Ambiguity in the specification may impair the ability to detect defects.
Bug-Reports	Demonstrates some degree of generalisation to unseen bug reports and realistic failures.	Focused mainly on GUI-related failures and informal bug descriptions. Further investigation is needed to understand how additional artefacts affect defect detection.

2.2.2 Code-Based Oracle Generation

Code-based oracle generation is an approach in which code or code-related documentation elements are used as the problem context alongside an LLM. In this section, the considered frameworks and tools, despite some use of LLMs for prompting strategies, serve as the main source of information, code, and/or code-related artefacts.

2.2.2.1 Software Under Test Documentation

This approach uses Software Under Test (SUT) artefacts, such as method descriptions and developer annotations, to infer expected program behaviour. For example, Javadoc comments, although not part of the executable code, often provide highly descriptive information about a method's intended functionality and can therefore be used to derive test oracles. The core motivation behind this line of research is that this type of code-related documentation can often be leveraged alongside the source code to achieve better algorithm contextualization. By combining these textual artefacts with contextual code information, researchers aim to reduce the oracle problem while minimising additional effort from developers. In this context, SUT artefacts are not limited to

natural-language documentation (e.g., Javadoc comments) but also include executable contextual elements, such as the focal method, test prefix, and partial test code, that convey implicit behavioural intent.

Early work in this area demonstrated that structured documentation, particularly Javadoc comments, can be systematically transformed into executable specifications. For example, techniques such as Jdoctor [3] demonstrate that natural-language documentation can be normalised and parsed to generate executable Java expressions that reflect expected behaviour. Comparative evaluations against previous documentation-based tools, such as Toradocu and @tComment, indicate that using richer code-related documentation artefacts can lead to more accurate oracle inference [18]. These results highlight the potential of SUT artefacts as a reliable source of behavioural information when they are well-written and consistent. Subsequent research expanded this idea by integrating documentation with execution context, including the method under test (MUT), its signature, and partial test code. Rather than generating assertions freely, frameworks like TOGA [8] approach oracle generation by ranking a small set of highly likely assertion or exception candidates. This design choice improves precision and helps ensure that the generated oracles are executable and relevant. The results of benchmarks such as Defects4J show that combining documentation with the test prefix context can significantly improve defect detection compared to approaches that use only documentation or code in isolation [8].

More recent studies emphasise the role of large language models in interpreting documentation to generate oracle responses. The research exemplified by Doc2OracLL [12] demonstrates that LLMs, when fine-tuned on documentation artefacts such as Javadoc comments, can generate accurate and strong test oracles. In particular, these studies reveal that documentation-driven approaches can sometimes outperform models that rely exclusively on the method under test, reinforcing the idea that comments encode behavioural expectations not explicitly present in the source code [12].

Another important direction within this approach focuses specifically on assertion generation rather than the full test. The underlying assumption of this methodology is that developers already define meaningful test setups; the main challenge shifts into formulating correct assertions. Approaches such as AsserT5 [12] show that providing LLMs with both the focal method, the test method context, and additional assertions consistently improves the quality of generated assertions across multiple metrics. However, comparative evaluations also reveal a disadvantage: despite providing richer context, it does not necessarily yield superior defect detection performance, as evidenced by weaker benchmark results compared to techniques such as TOGA [8, 25]. In this particular study, it is evident that the initial promising results are contingent on the context itself, indicating that there is little generalisation in this case.

Additionally, prompt engineering can be introduced alongside SUT documentation to provide more detailed context about the problem at hand. An example of this is AugmenTest [16] which, that constructs structured prompts that combine class-level information, method signatures, developer comments, and partial test code to guide oracle generation. Such an approach is also able to deliver promising results, shown in the experimental results, this however brings about two

major issues. The first concern of the authors is the applicability of this documentation-based approach, given the lack of projects rich in this type of information [16]. Lastly, due to the focus on Java projects, the researchers noted that framework applicability and performance might decline in other programming languages. An additional advantage of this approach is its model-agnostic nature, enabling its application across different LLM architectures.

A culmination of all the different methodologies within this approach was shown in a study that explores six prompt combinations, that used SUT documentation in order to assess what would be the most suited elements, and to understand if some artefacts could be leveraged whenever another is missing, and the results provided by TOGLL [11]. Researchers effectively identified which combinations could be tested, for example, only the test prefix or the test prefix alongside MUT documentation strings. It was shown that this approach could outperform previous similar approaches. Not only making it able to effectively detect defects, but also with a high degree of generalisation. This can be seen in the results of the tool's evaluation.

Regarding the effectiveness of the results achieved by SUT artifacts there was also a recent study [31] that explored how the test prefix and the focal method can be used, as input, and to which degree they are useful and effective at generating assertions and how viable is a code centric oracle. The provided input accurately predicted assertions with a high degree of certainty [31]. When evaluating on a known benchmark, this approach identified a respectable amount of defects, contributing to the acknowledgement of this methodology [31]. Lastly, there was an in depth evaluation on how the number of candidate assertions, assertion types, focal-test and assertion length, and LLM-Based approach can influence the results of the generated suite and it was identified that LLMs achieve higher prediction accuracy with multiple candidate assertions, some assertion types LLMs can predict with a higher degree of accuracy, fine-tuning LLMs will inherently achieve better results and lastly the focal-test and assertion lengths highly influence the prediction accuracy of LLMs [31]. The evaluation results indicate that SUT artefacts can effectively contribute to oracle generation, and this research is important to this topic. Despite the impressive results, this particular study was evaluated only using Java and JUnit; we cannot assume that the results are generalizable to other programming languages.

To conclude, giving SUT artifacts alongside some structured prompting can be leveraged to enhance the oracle generation, and the aforementioned approaches demonstrate the promising results of this line of research, the downside however is that with this methodology we are highly dependent on the quality of the artifacts of a given project, and having a real-world scenario in mind, it could be difficult to apply this approach for that exact reason.

Table 2.3: Software Under Test Documentation Oracle Generation Approaches

Tool	**Strengths**	**Limitations and Research Opportunities**
Jdoctor	Transforms Javadoc comments into executable specifications and derives expected program behaviour from documentation. Demonstrates improved oracle inference compared to earlier documentation-based techniques.	Strongly dependent on documentation quality and consistency. Performance may degrade when documentation is incomplete, ambiguous, or poorly maintained.
TOGA	Combines documentation, focal method context, method signatures and test prefixes to rank highly probable assertions and exceptions. Demonstrates strong defect detection performance on benchmarks such as Defects4J.	Restricted to a small candidate assertion space and evaluated mainly in Java ecosystems, raising concerns about applicability to other languages and frameworks.
Doc2OracLL	Fine-tunes LLMs on documentation artefacts (e.g., Javadoc) to generate strong test oracles. Can outperform approaches relying only on source code by exploiting behavioural information encoded in comments.	Highly dependent on rich documentation availability and quality. Generalisation to poorly documented systems remains uncertain.
AsserT5	Improves assertion generation by combining focal methods, test context and existing assertions. Produces high-quality assertions across multiple evaluation metrics.	Higher contextual richness does not necessarily translate to superior defect detection. Results indicate limited generalisation compared to approaches such as TOGA.
AugmenTest	Uses structured prompting that combines class-level information, method signatures, comments and partial test code. Model-agnostic and capable of improving the quality of oracle generation.	Dependent on projects with high-quality documentation and evaluated mainly on Java systems, limiting confidence in cross-language applicability.

Continued on next page

Tool	**Strengths**	**Limitations and Research Opportunities**
TOGLL	Explores multiple prompt combinations of SUT artefacts and identifies effective combinations when some artefacts are missing. Demonstrates strong defect detection and promising generalisation.	Still dependent on documentation and contextual artefacts availability, which may reduce practicality in poorly documented real-world projects.
Code-Centric Oracle Generation	Demonstrates that focal methods and test prefixes can accurately predict assertions and detect defects with good certainty. Also provides insights into factors influencing prediction accuracy (assertion type, focal-test length, fine-tuning).	Evaluated primarily on Java and JUnit, making it unclear whether results generalise to other languages, frameworks or testing ecosystems.

2.2.2.2 Prompt Based

Prompt-based variants are approaches that leverage the intrinsic benefits of LLMs, such as prompt engineering, structured prompting, chain-of-thought reasoning, or constructed self-repair mechanisms. Additional context can be added; however, the core of this approach is applying the advantages of LLMs.

A very creative approach is to use the aforementioned self-repair mechanism, which can be applied, for example, in ChatUnitTest [6] in two steps. The first is a rule-based repair mechanism that states two major rules developed by the research team. Whenever these two fail, the second step will take place: an LLM-based repair mechanism that is used to prompt the error at hand and, within a certain number of attempts, check whether it is possible to overcome such an obstacle; if the test is discarded. Despite this interesting approach related to self-repairing tests, there are more techniques used within this framework, such as the use of chain-of-thought "based on the dependency graph constructed during preprocessing " [6], an adaptive focal method context and a prompt template to construct the input prompt for the LLM. Despite the promising results in the evaluation section of this framework, there is still a question about its generalisation, since there is no evaluation on a known benchmark; therefore, further studies could be conducted to address this matter.

The same self-repair and feedback-driven prompting mechanism can be applied more systematically by combining static and dynamic analyses to support different phases of test oracle generation. In this context, static repair is primarily used to address compilation errors, while dynamic repair leverages runtime execution feedback to correct semantically invalid assertions. A representative example of this methodology is ChatAssert [10]. ChatAssert follows a two-phase

workflow consisting of a generation phase and a repair phase. In the first phase, candidate oracles are produced leveraging LLMs and carefully constructed prompts. The repair phase is then responsible for iteratively fixing oracles that either fail to compile or fail during execution. Unlike earlier approaches, ChatAssert integrates this repair process directly into the prompt engineering loop. A defining characteristic of ChatAssert is its strong reliance on prompt-engineering techniques. This can be identified in the generation repair step, and more precisely in the first two sub-phases, code summarisation and examples. Through multiple prompting iterations, the LLM is first provided with semantic summaries of the relevant classes and methods that appear in the test prefix. Finally, additional relevant code snippets similar to the problem at hand and within the project's context are added to the prompt for a few-shot learning approach. The final evaluation shows that the results are promising. Nevertheless, it shows concerns regarding applicability. In particular, the iterative interaction with large language models, combined with repeated compilation and test execution, incurs substantial computational and time overhead, and the evaluation is conducted exclusively on Java projects and JUnit-style assertions, raising concerns about its applicability to other programming languages. To conclude, this approach has identified some interesting cases, leveraged the benefits of LLMs, and yielded promising results. However, there is still a major concern regarding generalisation and real-world applicability, since these frameworks are predominantly tailored to the problem context and are computationally expensive and time-consuming.

Table 2.4: Prompt-Based Oracle Generation Approaches

Tool	**Strengths**	**Limitations and Research Opportunities**
ChatUnitTest	Combines self-repair, chain-of-thought reasoning, adaptive focal method context and prompt templates to improve generated tests. Includes a rule-based and LLM-based repair mechanism for failed tests.	Lacks extensive validation on established benchmarks, making generalisation uncertain. Framework behaviour may be overly tailored to the evaluated context.
ChatAssert	Integrates static and dynamic repair with iterative prompt engineering, semantic summarisation, and few-shot learning to improve oracle correctness. Demonstrates promising results in generation and repair workflows.	Computationally expensive and time-consuming due to repeated LLM interactions, compilation and execution cycles. Evaluated primarily on Java and JUnit, limiting evidence of applicability to other ecosystems.

2.2.3 Hybrid-Based Oracle Generation

Hybrid-based oracle generation leverages the known advantages of LLMs, such as code comprehension, alongside traditional software testing techniques. To be more concrete, these approaches

leverage LLM-specific benefits, such as structured prompting, prompt engineering, and chain-of-thought reasoning, while integrating traditional testing strategies, including white-box and black-box testing, mutation analysis, and static analysis techniques [7, 29, 30]. By leveraging both of these methodologies, hybrid approaches aim to address the limitations of purely LLM-driven or purely traditional testing methods. Recent studies highlight the effectiveness of a logic-driven hybrid approach in improving both the coverage and correctness of generated test suites [29]. In particular, incorporating white-box testing strategies, such as execution path coverage and condition coverage, alongside black-box techniques, including boundary value analysis and equivalence partitioning, enables a more thorough comprehension of program behaviour. When combined with a chain-of-thought prompting framework, these strategies provide LLMs with richer testing context and explicit coverage objectives, therefore reducing scenario omissions and improving defect detection capabilities [29].

A representative example of this approach is TestLoter, which demonstrates that leveraging LLM-based oracle and test generation through a logic-driven reasoning framework can significantly outperform traditional search-based tools such as EvoSuite [29]. The empirical results reported in TestLoter confirm that hybrid approaches can achieve superior code coverage and higher defect detection rates, reinforcing the potential of combining LLM reasoning with classical software testing principles.

In another study, static analysis was combined with chain-of-thought (CoT) reasoning to mitigate the limitations of LLM-based oracle generation and improve the overall quality of generated tests [30]. IntelliUnitGen "synthesises static code features extracted via static analysis " [30] such as class and method signatures, control-flow structures, decision nodes, exception handling paths, and class dependencies, and this information is then converted into structured prompts that are used in a CoT prompting strategy. IntelliUnitGen also employs a coverage-aware prompting template that targets multiple coverage criteria, including statement, branch, condition, decision, and path coverage. This enables LLMs to have better context for the solution, avoid missing-context scenarios, and improve oracle correctness. Evaluation done on real-world Java projects identifies that IntelliUnitGen is able to achieve better performance, in comparison to static analysis-based tools as well as LLM-only approaches, in terms of coverage, generalisation capabilities, compilability and executability [30]. These results not only identify the robustness achieved when combining static analysis with CoT, but also the generalisation, overall coverage, and defect detection of hybrid approaches.

Mutation testing can also be leveraged alongside a prompt augmentation approach to assist with oracle generation and produce a more thorough test suite with a high defect detection rate. A recent study that follows this methodology is MuTAP [7], which integrates LLMs with mutation analysis within an iterative prompt-augmentation framework. In its initial stage, MuTAP employs a zero- or few-shot prompting strategy, using only the program under test to generate the initial set of unit tests and associated oracles. To ensure the correctness of the generated oracles, the approach also includes a repair phase related to syntactic errors. The key innovation of MuTAP is the use of mutation analysis to guide prompt construction. The first step of this framework,

as stated previously, is to generate an initial test suite based on the program under test, followed by the repair phase. In the next step, mutation testing is applied to identify surviving mutants. These surviving mutants are then incorporated directly into the prompt to given additional context about what still needs to be covered by the existing test suite, this process will occur until little to no mutants are left [7]. To conclude by iteratively augmenting the prompt with such mutation feedback, MuTAP guides the LLM to generate new test cases and/or oracles that specifically target previously undetected faults.

In conclusion, with Hybrid Based approaches, we are capable of leveraging various artefacts and tools, and making use of traditional methods alongside LLMs have been proven to not only help for generalisation purposes, meaning that the approaches can be used in different contexts, but also in the ability to make a test suite that is able do detect defects. There are still some artifacts that could be used in further research to understand what other type of information could be leveraged alongside LLMs and traditional methods, an interesting topic would be how product specifications could influence the oracle generation.

Table 2.5: Hybrid-Based Oracle Generation Approaches

Tool	**Strengths**	**Limitations and Research Opportunities**
TestLoter	Combines LLM reasoning with classical testing strategies such as white-box and black-box testing. Demonstrates improved coverage and defect detection compared to search-based tools like EvoSuite.	Relies on carefully designed logic-driven prompts and structured testing objectives, which may limit adaptability to less structured testing environments.
IntelliUnitGen	Integrates static analysis (e.g., control-flow, method signatures, dependencies) with chain-of-thought prompting. Improves coverage awareness and generates more contextually accurate test oracles.	Strong dependence on static analysis quality and Java-specific tooling may limit generalisation to other programming languages and ecosystems.
MuTAP	Uses mutation analysis to iteratively refine LLM-generated test suites. Mutation feedback is incorporated into prompts to progressively improve defect detection and test completeness.	Computationally expensive due to iterative mutation testing cycles. Performance and scalability may be affected in large software systems.

2.2.4 Complementary Work

In the context of oracle generation, there are articles that contribute to the current body of knowledge by reviewing the state of the art [4] and categorising and explaining the different schools

of thought. Other research contributes a new metric for evaluating the quality of generated oracles [21]. This section will explore the aforementioned contributions.

Within the context of oracle generation there are articles that that contribute to the current body of knowledge by having a review of the current state of this topic [4], categorizing and explaining the different schools of thought. In the context of oracle generation, there are articles that contribute to the current body of knowledge by reviewing the state of the art [4] and categorising and explaining the different schools of thought. Other research contributes a new metric for evaluating the quality of generated oracles [21]. This section will explore the aforementioned contributions.

Recent researchers made a comprehensive review of large language models for automated test case generation [4]. This particular study focuses on an, in depth, analyses on research published between 2022 and 2025, making a categorization of LLM based test generation approaches. Contrary to other research, the authors not only identify and classify test generation methods, but also examine the most frequently used datasets as benchmarks and the language models used. This review identifies trends in the most common approaches to test generation, outlines current limitations such as inconsistent test correctness and compilation errors, and mentions future challenges related to scalability, concerns about the evaluation of results, and, lastly, the applicability in a real-world scenario. This research serves as an important reference for understanding the current research landscape, its challenges, and potential future research gaps.

Regarding metrics for evaluating oracle quality, State Field Coverage has been suggested [21]. The main idea behind this metric is that the quality of an oracle can be measured by the thoroughness of inspection of the internal state of the objects under test. In other words, an oracle that covers a larger portion of an object's internal state is more likely to detect faults than one that fails to do so. The suggested framework takes a target class as input and internally constructs a type graph representing the object's state, including its fields and the relationships between their types. Based on this graph, the framework identifies the total number of state fields that can be examined and those actually examined by the generated oracle. This type of analysis is comparatively cheaper than, for example, mutation analysis, which is computationally more expensive. Evaluation using this metric also demonstrates a strong correlation with the mutation score [4]. Other experiments conducted on the Defects4J benchmark show that prioritising oracles with higher state-field coverage enables real faults to be triggered with significantly fewer tests.

2.2.5 Online Learning Platforms

Although not directly related to the Oracle Problem, online learning platforms have attracted significant research attention in recent years. Existing studies have explored a variety of dimensions, including learning effectiveness and the extent to which users achieve the intended educational outcomes [13],[5], platform usability [14],[23], accessibility [28],[2], dropout rate prediction [27], and methods for assisting learners in selecting the most suitable platform [24].

Despite this growing body of research, little attention has been devoted to the reliability of the automated feedback mechanisms employed by these platforms. In particular, the literature provides limited evidence regarding whether solutions accepted by online programming learning

Table 2.6: Complementary Work in LLM-Based Oracle Generation

Contribution	Strengths	Opportunities for Improvement
LLM Test Generation Review (2022–2025)	Provides a comprehensive categorisation of LLM-based test generation approaches. Analyses datasets, benchmarks, and models, while identifying key trends and limitations such as inconsistent correctness, compilation errors, and scalability challenges.	Limited to studies within a specific time window (2022–2025), which may exclude earlier foundational work. Evaluation inconsistencies across surveyed studies make cross-comparison difficult.
State Field Coverage Metric	Introduces a novel metric for evaluating oracle quality based on the extent of internal object state inspection. Demonstrates correlation with mutation score and effectiveness in detecting real faults with fewer tests. More computationally efficient than mutation-based evaluation.	Focuses primarily on internal state visibility, which may not fully capture external behavioural correctness. Applicability to complex or highly dynamic systems remains to be further validated.

platforms are always consistent with the corresponding problem specifications. Given the central role that automated feedback plays in the learning process, this represents an important research gap that warrants further investigation.

2.2.6 Discussion

Among the various approaches mentioned previously, it is important to understand what should be done to extend this line of research.

Having only code as the main focus for oracle generation, despite different prompting strategies, has proven to be efficient to some extent, yielding promising results in some cases. This, however, should not be the sole source of context for an oracle, since we should not assume that the implementation is always correct, thereby compromising the school of thought that only leverages code for this type of task, as analysed by recent research [20]. Leveraging code and code-related documentation will increase the quality of the generated oracle, as shown in the related section. Despite these results, generalisation remains a concern due to the tailored prompting strategies used, which can result in an oracle suited to a particular type of problem. An important element described in ChatUnitTest [6] is the self-repair mechanism that assists in generating the test suite. This can be employed in future research to improve the compilability of the generated tests and thereby contribute to an overall better test suite. Another important aspect of a code-based approach is leveraging code-related artefacts, as they provide additional context, for example, from [15]. This approach, however, can, in some cases, suffer from a lack of flexibility in how we use this type of documentation, as seen in Jdoctor [3], but made more manageable on

TOGA [8]. Recent research shows that this type of context should be leveraged for oracle generation, as it not only provides the source code of the method under test but also additional context, thereby overcoming the initial challenge of relying solely on code as context for oracle generation.

Leveraging context-related documentation is another approach mentioned previously. This, however, has some inherent difficulties and opportunities. Bug-related documentation, such as bug reports, offers a compelling approach to this problem, and the work in ANDROB20 [15] demonstrates that an effective test suite can be generated from such artefacts. Leveraging requirements, whether formal or informal, is a different subject, as this approach has inherent difficulties. Having lengthier and more complex requirements will lead to worse performance, coverage-wise [17], due to LLMs' limited ability to significantly impact overall coverage, especially when handling lengthy text descriptions with nested conditions. It has been shown that this will significantly impact overall coverage, especially under more complex coverage criteria [17]. Future research should leverage this type of specification carefully.

Hybrid approaches have shown that blending prompting strategies with traditional testing strategies is beneficial for oracle generation and generalisation [7, 29, 30]. It has been identified that this combination will not only contribute to a higher degree of generalisation but also contribute to the defect detection capability, which is ultimately what is desired from an oracle. There are opportunities in this topic, and future research can explore to what extent different types of specifications could be used to achieve the best generalisation and defect detection rates.

Finally, an interesting research opportunity emerges from the intersection between oracle generation and online programming learning platforms. While the literature has extensively explored techniques for improving the quality of generated test suites and/or oracles, little attention has been devoted to using these approaches as independent mechanisms for assessing the reliability of the feedback provided by educational platforms, or even to improve the platform test set for a more consistent problem's solution validation. Given that learners often rely on automated evaluation systems as indicators of correctness, investigating whether accepted solutions truly conform to their specifications represents a relevant and largely unexplored research direction. Consequently, combining oracle generation techniques with formal verification approaches may not only advance software testing research but also yield valuable insights into the trustworthiness and robustness of automated assessment systems used in programming education.

2.3 Research Gap

There is a clear opportunity to further investigate hybrid approaches to test oracle generation that combine natural-language problem specifications with complementary artefacts and advanced prompting strategies. In particular, it remains unclear how artefacts such as preconditions, post-conditions, and formal proofs can be effectively integrated with specification-based test generation to improve the identification of incorrect implementations. Such integration has the potential to increase confidence in the validation process by enabling multiple, complementary perspectives on software correctness. Additionally, it is relevant to explore whether self-repair-inspired

methodologies can be used to iteratively improve the quality of generated test suites by addressing limitations observed in earlier execution phases.

A second, closely related research opportunity concerns the evaluation context of these approaches. Existing studies have primarily focused on traditional software engineering settings and benchmark datasets, with limited attention given to online academic programming platforms. These platforms play an important role in programming education by providing automated feedback to large numbers of learners. However, the correctness and reliability of this feedback are typically assumed rather than systematically validated. This raises the question of whether solutions accepted by such platforms are always consistent with their intended specifications and whether additional validation mechanisms can reveal hidden inconsistencies.

Overall, there is a lack of empirical studies that evaluate the combined use of specification-based test generation, formal specification derivation, formal verification, and prompt refinement within a unified workflow. Addressing this gap would contribute to a deeper understanding of the benefits and limitations of integrating these techniques, including trade-offs in effectiveness, robustness, and generalizability. Furthermore, applying such a unified workflow to online programming learning platforms may provide valuable insights into the reliability of their automated evaluation systems. Given learners' reliance on this feedback, improving the trustworthiness of these systems is essential to supporting effective programming education and ensuring the validity of assessed solutions.

2.4 Final Considerations

This chapter reviewed the relevant literature for this dissertation, with particular emphasis on recent developments in the Oracle Problem in the context of Artificial Intelligence. It discussed how the emergence of Large Language Models has influenced automated test generation and oracle construction, as well as how these techniques can be applied within online programming learning platforms.

Based on this review, several research gaps were identified, particularly concerning the reliability of platform feedback and the combined use of specification-based test generation, formal reasoning, and prompt engineering techniques. The following chapter presents the proposed approach designed to address these gaps.

Chapter 3

Proposed Approach

This chapter presents the workflow of the suggested framework to help us determine whether we can indeed trust the feedback provided by Academic Online Platforms. To establish the necessary context, the problem this research addresses is first introduced and discussed. Subsequently, each phase of the proposed workflow is presented in detail, highlighting its objectives, inputs, outputs, and role within the overall methodology.

3.1 Problem Contextualization

Although previous research has explored the use of problem specifications as test oracles, these approaches are often evaluated in isolation and within controlled experimental settings. Consequently, there is still limited understanding of how specification-based test generation can be combined with complementary techniques, such as formal specification derivation and formal verification, to improve confidence in the validation process.

The problem addressed in this dissertation is therefore twofold. First, it is necessary to determine whether problem specifications can effectively support the identification of incorrect implementations when used as the basis for automated test generation. Second, it is important to understand whether additional artefacts, namely preconditions, postconditions, and formal proofs, can complement generated test suites and serve as a warning mechanism for detecting potential inconsistencies.

To investigate these questions, this study applies a unified workflow within the context of an online programming learning platform. In this environment, accepted solutions are generally assumed to be correct, despite the limited evidence regarding the reliability of the underlying evaluation mechanisms. By combining specification-based test generation, formal verification, and prompt refinement strategies, this dissertation seeks to evaluate both the effectiveness of the proposed methodology and the trustworthiness of the feedback provided by such platforms.

3.2 Framework Workflow

To conduct the proposed analysis, a five-step workflow is defined. The first three phases constitute the core pipeline: (1) collection and verification of problem solutions and their corresponding specifications, (2) generation of test cases based on problem specifications, and (3) derivation of preconditions and postconditions to construct a formal specification. Together, these steps establish the foundation for evaluating specification-based test generation and formal verification.

Two additional phases extend this core workflow. These are not strictly part of the initial pipeline, but are introduced to assess the generalisability of the approach and the influence of prompt design on the test case generation results. Specifically, Phase (4) evaluates the performance of the initial prompt across different problem categories, while Phase (5) investigates prompt refinement and its impact on test generation quality. Collectively, these phases enable a broader evaluation of both methodological robustness and prompt sensitivity.

All experiments were conducted using the `openai/gpt-oss-120b` model[1], selected as a balanced choice between generation quality and computational cost [22], given the scale of the study, which spans 207 problems across multiple generation tasks.

The following sections provide a detailed description of each phase of the workflow. Furthermore the proposed methodology can be found in a public available Github Repository.[2]

3.3 Phase 1: Retrieval and Verification

The objective of this phase was to construct a dataset containing both problem specifications and corresponding solutions, while ensuring that all collected implementations were accepted by the target platform.

To obtain the solutions, two public GitHub repositories were used as primary sources. A custom script was then employed to retrieve the corresponding problem specifications from the platform. For each challenge, the script collected the problem description, input and output specifications, associated constraints, and the examples provided by the platform. All retrieved information was stored for subsequent phases of the study.

Some problem statements included images in their descriptions. Since the focus of this work was on textual specifications and no scalable image-processing strategy was available within the scope of the dissertation, these images were replaced with detailed textual descriptions whenever necessary.

The phase was composed of three main activities:

1. Retrieval of solutions from two public GitHub repositories. Although the repositories contained solutions across multiple categories, only challenges in the selected category were considered at this stage of the study.

[1] https://onyx.app/open-llm-leaderboard
[2] https://github.com/Andre-Pestana0/Educational-Platform-Analysis

2. Retrieval of the corresponding problem specifications, including descriptions, constraints, and input/output examples.

3. Verification of the collected solutions through automatic submission to the platform in order to confirm their acceptance status. If the solutions were rejected by the platform these would be corrected and verified until accepted.

The verification process revealed that the vast majority of the collected solutions were accepted without modification. Of the 207 Python solutions in the dataset, only three (Challenges 1237, 1273, and 2167) required correction before the subsequent phases could be executed.

Figure 3.1 illustrates the workflow followed during this phase.

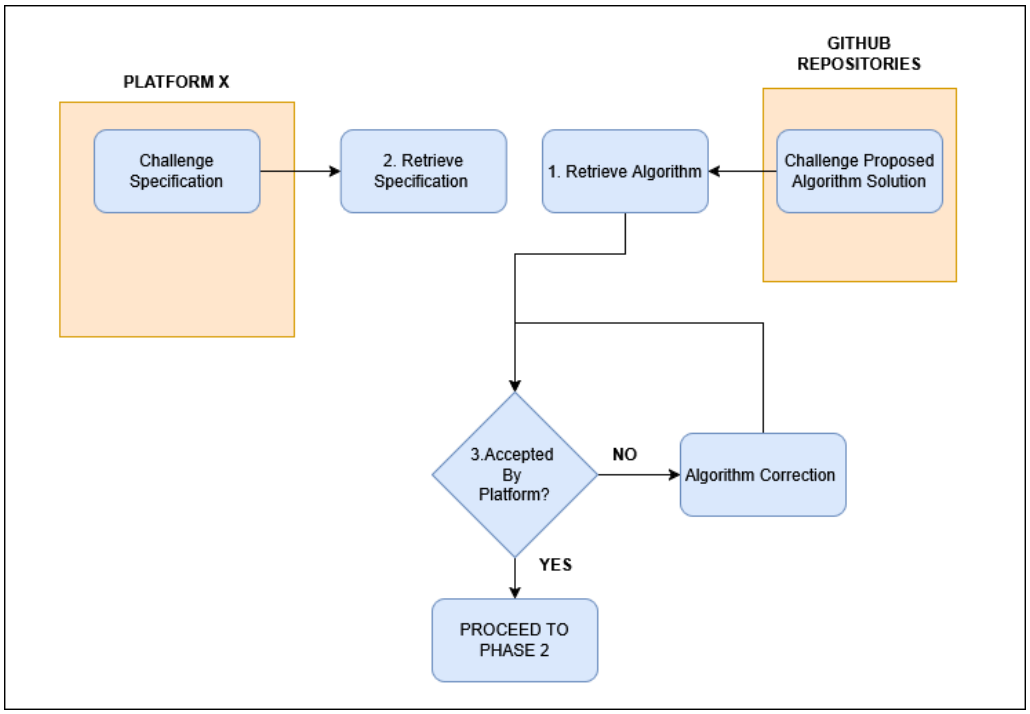


Figure 3.1: First Phase of the Methodology.

3.4 Phase 2: Test Case Generation

In this phase, problem specifications are treated as test oracles and leveraged to generate test suites for each programming challenge. To support this process, a structured prompt was designed to guide the Large Language Model (LLM) in producing comprehensive test cases that are compliant with the problem specifications (see Figure 3.2). Beyond the generation of test cases, this phase also serves an exploratory objective: to evaluate the extent to which natural language problem specifications can effectively function as test oracles and if the generated test cases are able to assist to evaluate the learning platform feedback. The inherent limitations and advantages of this approach will be identified.

This phase is composed of two main activities. First, the specification of each exercise is provided as input to the LLM, which generates a corresponding set of test cases. These test cases are then extracted and stored for subsequent analysis. Secondly, the generated test cases undergo a validation process in which the LLM-proposed inputs and outputs are verified against the solution, i.e., the solution is executed with the LLM-suggested inputs, and the outcome is directly compared to the LLM-suggested output. If there are no compilation errors and both outputs match, the test case is considered valid; otherwise, further analysis is needed.

This verification ensures not only that the generated inputs conform to the problem constraints and can be successfully executed by the solution, but also that the outputs predicted by the LLM match those produced by the reference implementation when executed with the generated inputs. As will be discussed in the following section, some generated test cases did not meet these requirements, highlighting the challenges of relying exclusively on natural language specifications for test generation.

Figure 3.3 presents a visual representation of the workflow implemented during this phase.

```
 You are acting as a strict black-box software tester.
You will receive a description of the programming problem.
Your task is to generate a comprehensive and non-redundant set of
test cases that fully exercise the problem specification.

  • Generate ALL useful and distinct test cases necessary to
    properly validate the problem.
  • Cover:
      – Normal cases
      – Edge cases
      – Boundary conditions
      – Minimum constraints
      – Maximum constraints
      – Special cases implied by the specification
  • Do NOT reuse, restate, or modify any example given in the
    problem.
  • Do NOT generate redundant or equivalent test cases.
  • Ensure inputs strictly match the input specification.
  • Ensure outputs are 100% logically correct.
  • Stop generating cases when additional cases would be redundant.
  • Problem description:  «Problem description»
  • Input:  «Problem Input description»
  • Output:  «Problem output description»
  • Examples:  «Problem examples»
```

Figure 3.2: Prompt used for test case generation from problem specifications

It is important to note that failing test cases, defined as cases in which the execution completes successfully but the expected output differs from the output produced by the reference solution, were individually inspected to determine whether the discrepancy originated from an incorrect test case or from a genuine defect in the solution. However, these manual inspections were not performed as part of the intended workflow. Instead, they were conducted exclusively for research purposes, namely to assess whether the subsequent phases of the proposed approach could detect issues already identified through manual analysis and to evaluate the degree of confidence that could be placed in LLM-driven automation throughout the process.

The long-term objective of the proposed workflow is to minimise human intervention. Ideally, manual verification would be required only when a warning signal is raised during the formal verification stage, indicating a potential inconsistency or defect. Such a level of automation is particularly important when considering the approach's scalability. While manual inspection may be feasible in a controlled experimental setting, applying the same process to larger collections of exercises or real-world software systems would quickly become impractical without automated support.

Consequently, conducting this study is essential for determining whether the proposed combination of LLM-generated artefacts and formal verification techniques can provide sufficient reliability to support a largely automated workflow. The findings of this investigation will help establish the extent to which manual validation can be reduced while maintaining confidence in the correctness of the generated results.

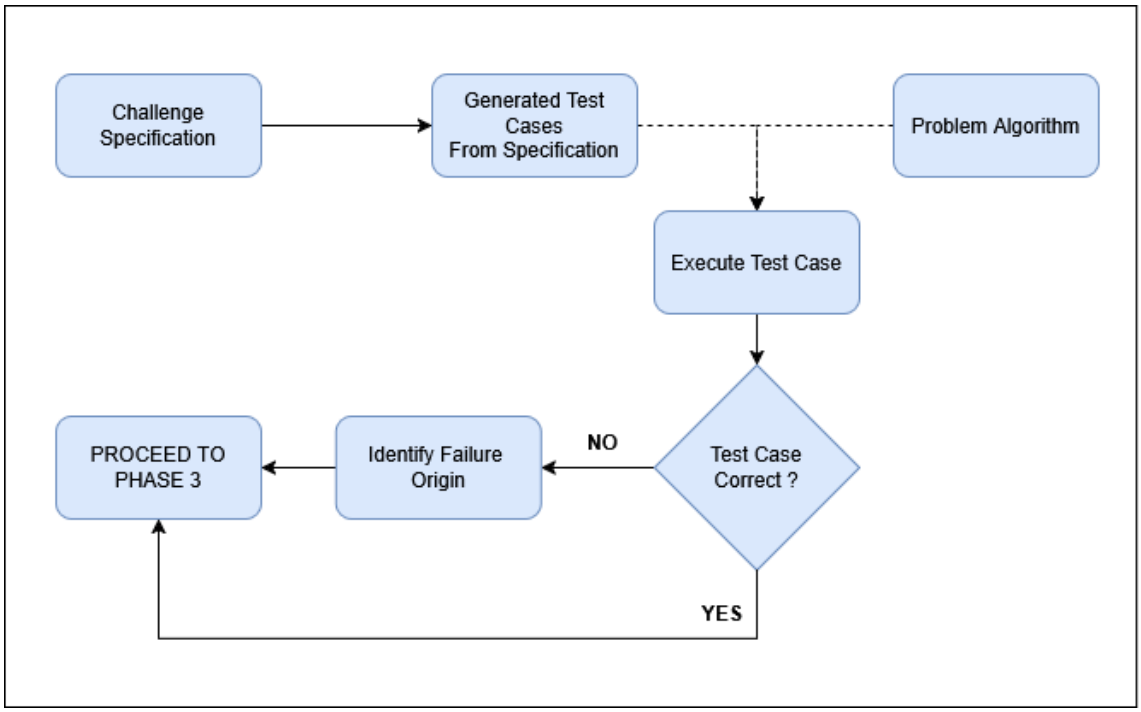


Figure 3.3: Second Phase of the Methodology.

3.5 Phase 3: Conditions and Formal Proof Generation

Formal verification techniques can be employed to identify anomalies and potential defects in algorithm implementations. To enable this process, it is first necessary to define a set of preconditions and postconditions for each exercise, as these serve as the foundation upon which formal proofs can be constructed. To facilitate the generation of these artefacts, two dedicated prompts were developed: one for generating the preconditions and postconditions from the problem specification (see Figure 3.4) and another for generating the corresponding formal specification and proof (see Figure 3.5).

```
Can you write, in Math notation, the formal specification
(just pre and post conditions used in Hoare logic) of this
natural language specified problem «Problem description»
```

Figure 3.4: Prompt used to generate pre- and post-conditions

```
Given the formal specification of the pre and post
conditions (as used in Hoare Logic) below, can you prove the
correctness according to the specification of the following P
«Programming language» program «Program»
```

Figure 3.5: Prompt used to generate pre- and post-conditions

This phase consists of two main steps. First, the preconditions and postconditions are generated from the natural-language problem specification as the primary source of information. Subsequently, the generated conditions, together with the corresponding algorithm implementation, are provided to the LLM to produce a formal proof of correctness. In cases where the generated proof was determined to be incorrect, i.e., the outcome did not align with the truth after a detailed analysis, the proof generation process was repeated. This evaluation process is discussed in the following chapter. A visual representation of this phase is provided in Figure 3.6.

This phase is particularly important because formal proofs provide a systematic mechanism for identifying potentially incorrect implementations. Rather than relying exclusively on generated test cases, the proof serves as an additional verification layer that highlights inconsistencies between an algorithm and its intended specification. The outcome of the verification process can classify an algorithm as “Totally Correct”, “Partially Correct”, or “Incorrect”. An example of a formal proof can be found in appendix A. A totally correct algorithm is one that satisfies its specified preconditions and postconditions and is guaranteed to terminate upon execution. A partially correct algorithm satisfies the specified conditions whenever it terminates, but termination itself cannot be guaranteed. Finally, an incorrect algorithm is one that fails to satisfy the specified requirements and therefore does not implement the intended behaviour.

It is important to emphasise that the generated formal proofs are not treated as absolute evidence of correctness. Since the proofs are produced with the assistance of an LLM, they remain susceptible to inaccuracies and reasoning errors, as will be demonstrated in the experiment chapter.

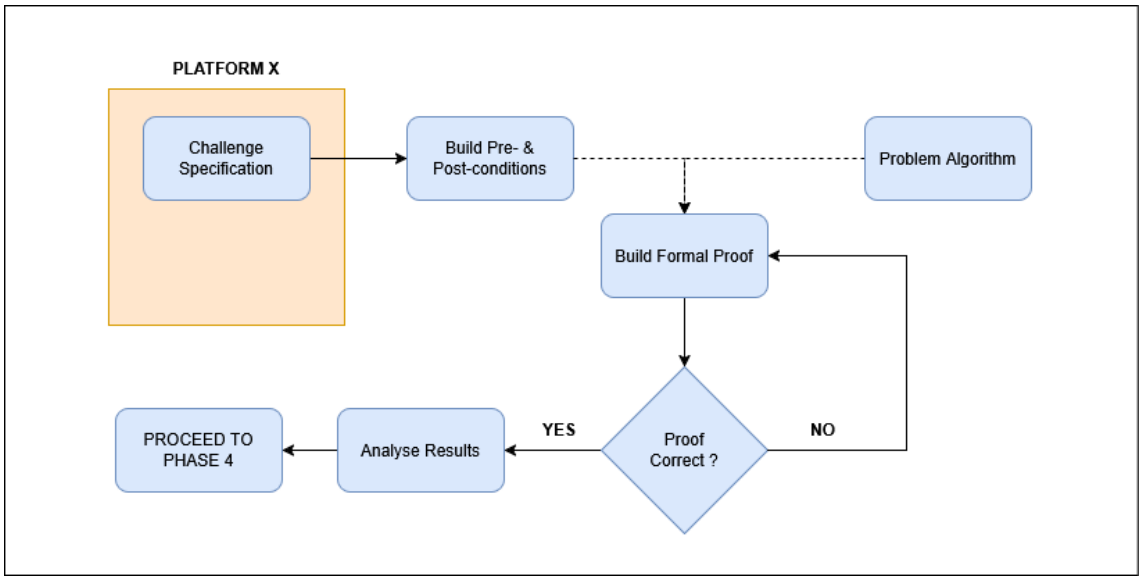


Figure 3.6: Third Phase of the Methodology.

Nevertheless, despite these limitations, the generated proofs provide valuable guidance by identifying implementations that are more likely to contain defects or inconsistencies. Consequently, they serve as an effective mechanism for prioritising further investigation and reducing the effort required for manual inspection.

To conclude, this phase aims to understand if formal proofs can provide assistance in identifying if the online platform is able to correctly distinguish correct from incorrect solutions.

3.6 Phase 4: Generalisation Evaluation

Following the detailed analysis of exercises from the "Iniciante" (Beginner) category, it is necessary to assess whether the same methodological approach, particularly the designed prompts, can be applied effectively to problems from other categories. This step is essential in order to determine whether the proposed methodology generalises across different types of programming challenges or whether the previously obtained results are primarily context-dependent and specific to beginner-level tasks.

To address this question, the first three phases of the proposed workflow are replicated across a selection of exercises from four distinct categories: "Strings", "Mathematics", "AD-Hoc", and "Data Structures and Libraries". For each category, three representative challenges are selected and processed using the same pipeline, including solution retrieval, test case generation, and formal specification-based verification.

This phase concludes with a comparative analysis of results across categories to evaluate the robustness, adaptability, and generalizability of the proposed approach beyond the initial experimental setting.

To conclude, these phases aim to demonstrate the degree of generalisation of the proposed framework.

3.7 Phase 5: Prompt Refinement Evaluation

In the final phase, the objective is to determine whether the results from the previous stages can be further improved by refining the base prompt and to assess whether the limitations identified in earlier experiments can be leveraged to refine the original prompting strategy. More specifically, this phase investigates whether prompt refinement can improve performance by increasing the Test Execution Success Rate and enhancing the detection of defective implementations.

To address these questions, a two-step process is followed. First, the subset of challenges that exhibited the most problematic behaviour, i.e., those with a 0% Test Execution Success Rate, is selected and re-evaluated by feeding the original prompt to *Claude Sonnet 4.6*. The issues identified in this context, as discussed in the following chapter, are then used to guide the refinement of the initial prompt. To perform this refinement iteration, the criteria used to halt this process is having a prompt capable of achieving a better Test Execution Rate (explained further in Chapter 4) than previously, resulting in only one prompt refinement iteration. The goal of this phase is to improve the quality of the generated test cases and assess whether the updated prompt better supports both correct test generation and defect detection.

Secondly, the refined prompt is applied to generate a new set of test cases for all problems that previously yielded failing results. Exercises that have already been identified as accepting incorrect solutions are excluded from this step, as they do not provide additional meaningful information for evaluating improvements in test-generation quality.

This workflow is illustrated in Figure 3.7.

Following the description of each phase, the next chapter presents a structured analysis of the results, difficulties encountered, and key insights gained throughout the study, aimed at addressing the defined research questions.

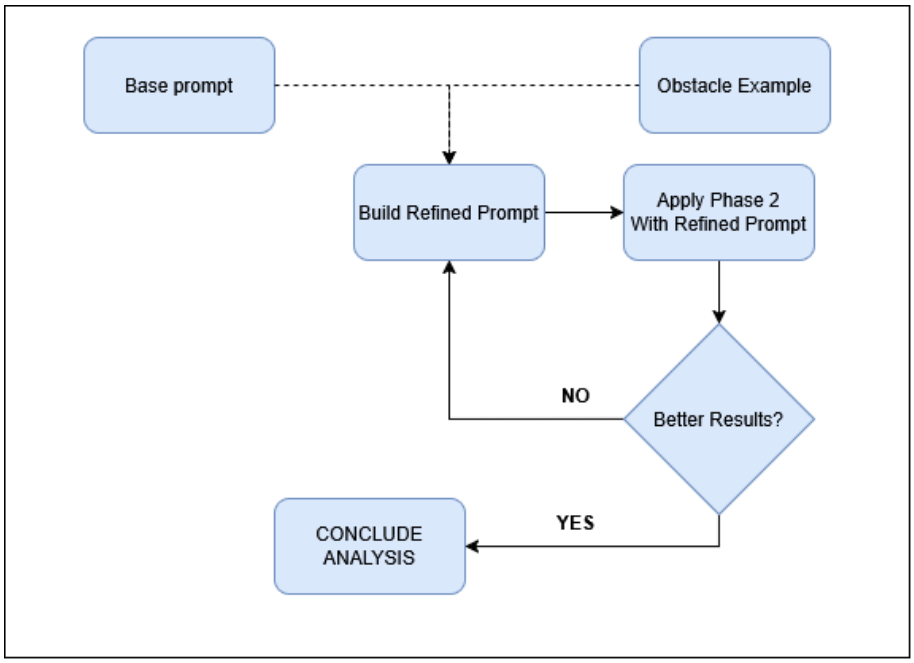


Figure 3.7: Fifth Phase of the Methodology.

Chapter 4

Experiment

This chapter presents a comprehensive analysis of each experimental phase. In addition to reporting the obtained results, it offers a critical reflection on the challenges encountered throughout the process, the adjustments made in response to these issues, and the key insights derived from the overall methodology. The aim is to provide a transparent and rigorous account of the experimental workflow from its initial design to its final evaluation. Finally, it aims to analyse the data that will respond of the research questions of this dissertation, namely:

- **RQ1:** Can we trust the feedback provided by programming learning platforms?
- **RQ2:** Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of test cases?
- **RQ3:** Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of formal proofs?
- **RQ4:** To what extent does prompt design influence the quality of LLM-generated test cases?

4.1 Platform Exploration

Before presenting the experimental study, it is important to provide an overview of the selected platform. To preserve anonymity, the educational online platform analysed in this dissertation will be referred to as *Platform X*. At the time of data collection, Platform X contained 2,411 programming challenges across nine distinct categories, each comprising between 50 and 853 challenges. The distribution of challenges per categories can be seen in Table 4.1.

The primary category considered in this study is *"Iniciante"* ("Beginner" in English), which the platform describes as containing "basic problems for anyone who has just started programming." Platform X is designed to support programming education and skills assessment through a collection of challenges with varying levels of complexity. Each challenge includes a problem specification and requires participants to submit a solution in one of several supported programming languages, including Python, R, Ruby, Rust, C#, C++, and Java, among others. For this

study, only implementations in Python will be considered, and a total of 219 challenges will be included, with 207 in the first three phases and an additional 12 in phase 4. Phase 5 will revisit 58 previously seen challenges.

Once submitted, solutions are automatically evaluated against a hidden set of test cases that are not publicly available. A solution is considered correct if it satisfies all the platform's validation criteria. Although Platform X does not provide official solutions for its challenges, it offers a discussion forum where users can exchange ideas, discuss common difficulties, and share potential solutions.

Each challenge is identified by a unique numerical code and is associated with a difficulty level ranging from 1 to 10. Within the Beginner category, however, challenges are only assigned difficulty levels between 1 and 9. The platform does not disclose the criteria used to determine these difficulty ratings, making it difficult to assess the factors that influence their assignment.

Table 4.1: Distribution of Problems by Category

Category	**Number of Problems**
Beginner	334
Ad-hoc	853
Strings	150
Data Structures and Libraries	180
Mathematics	269
Paradigms	215
Graph	277
Computational Geometry	83
SQL	50

4.2 Phase 1: Retrieval and Verification

As previously mentioned, this phase involves retrieving algorithm implementations and their corresponding problem descriptions. In total, 207 Python implementations were collected from a broader set of 341 available challenges. Although the majority of the selected exercises are classified with difficulty level 1 (see Figure 4.1), the dataset is not limited to beginner-level problems and includes challenges spanning a wide range of difficulty levels.

The distribution of difficulty levels is as follows: 120 challenges at difficulty 1, 51 at difficulty 2, 20 at difficulty 3, 9 at difficulty 4, and 4 at difficulty 5. Additionally, difficulties 6, 7, and 9 are each represented by a single challenge (see Figure 4.2).

Although the assigned difficulty level is not a definitive indicator of problem complexity or specification quality, it can nevertheless provide useful context for understanding whether higher-difficulty challenges tend to introduce additional challenges in test generation and verification.

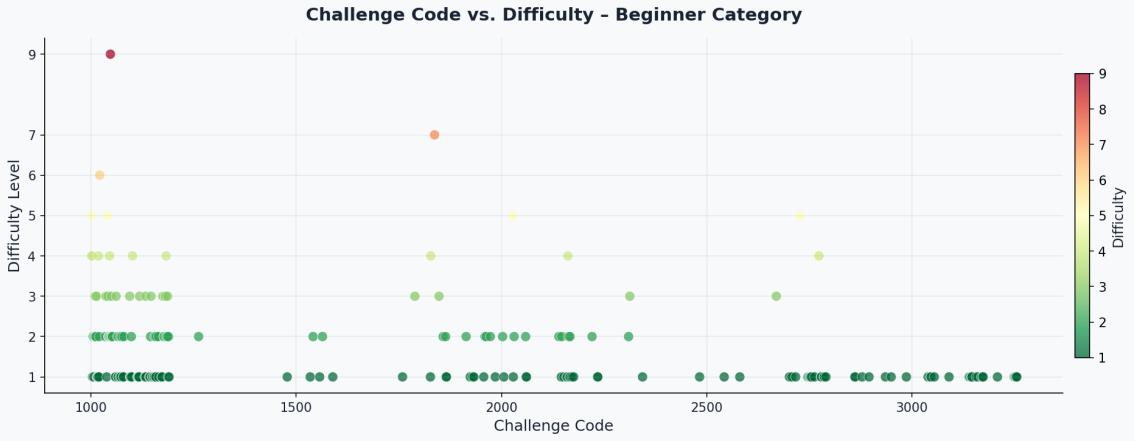


Figure 4.1: Difficulty distribution scatter graph

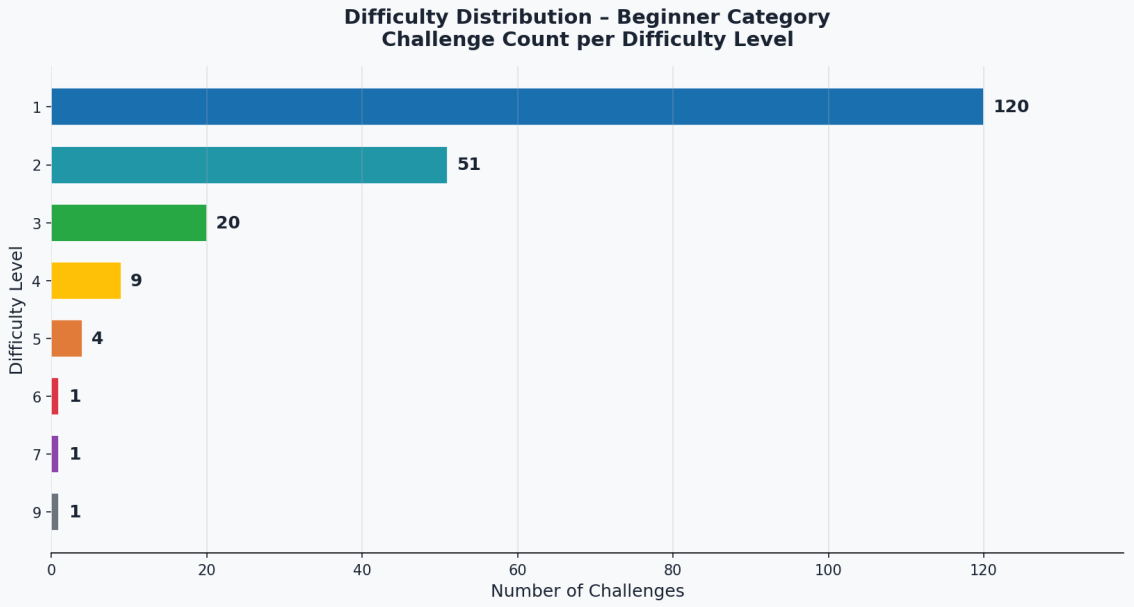


Figure 4.2: Difficulty distribution

In particular, it may help to assess whether more complex problems are associated with a higher likelihood of difficulties in constructing adequate test cases, or whether other factors play a more significant role in influencing the suitability of problem specifications as implicit oracles.

Regarding solution retrieval, the two utilised repositories presented minimal issues. Only three solutions, corresponding to challenges 1237, 1273, and 2167, required correction in order to be accepted by the platform (see Figure 4.3). This limitation did not affect the overall analysis, as the necessary adjustments were successfully resolved using an alternative large language model, distinct from those employed in the standard workflow.

To conclude, this phase focused on constructing the dataset used throughout the study and analysing the distribution of challenge difficulty levels across the selected exercises. The collected dataset revealed a clear concentration of challenges at difficulty level 1, indicating that the majority

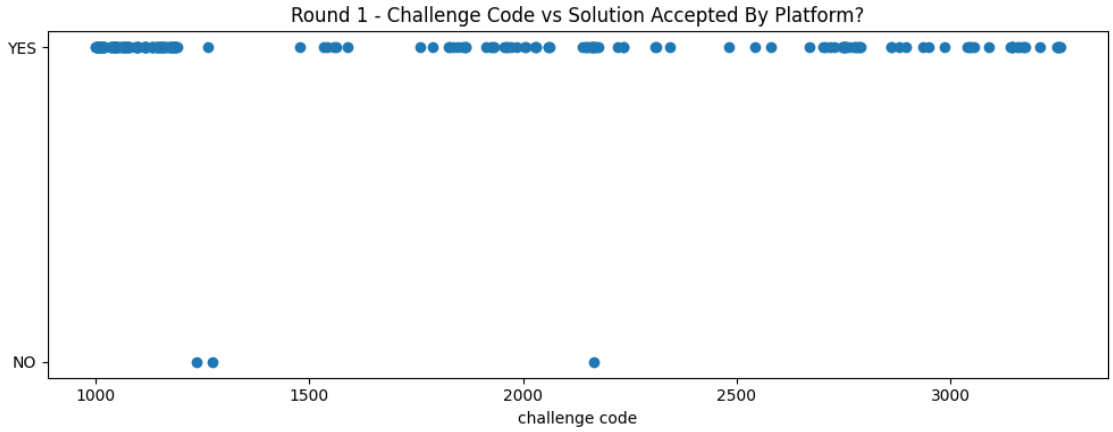


Figure 4.3: Solution acceptance

of the analysed problems are aimed at beginners.

Additionally, the verification process confirmed that the vast majority of the retrieved solutions were correctly accepted by Platform X. Only three solutions exhibited inconsistencies and therefore required correction before they could be included in the subsequent phases of the study. These corrections were performed using a different LLM, independent of those used in the experimental workflow, thereby preserving the integrity of the evaluation process.

4.3 Phase 2: Test Case Generation

Since this phase relies on problem descriptions as implicit test oracles, it provides an opportunity to evaluate the effectiveness of such artefacts in generating meaningful test suites. Across the 207 analysed challenges, a total of 1,954 test cases were generated, corresponding to an average of approximately nine test cases per challenge. However, the distribution is not uniform: some exercises contain as few as one generated test case, while others include significantly larger test suites, with one outlier reaching 78 test cases.

Overall, the distribution of generated test cases across difficulty levels appears relatively balanced, as illustrated in Figure 4.4. With the exception of a notable outlier in a difficulty level 2 exercise, which generated 78 test cases, most difficulty categories exhibit comparable distributions. This observation suggests that difficulty level alone is not a strong determinant of the size of the generated test suite. In other words, more complex problems do not necessarily lead to a higher number of generated test cases, nor do simpler problems consistently result in fewer tests.

This raises the question of what factors drive the observed variability in test suite size.

4.3.1 Variability in test suites

Figure 4.4 identified that difficulty alone is not enough to influence the amount of test cases generated by the LLM, since it would be expected to have fewer test cases for lower difficulty challenges and more for the more difficult ones. To better understand what other factors can influence

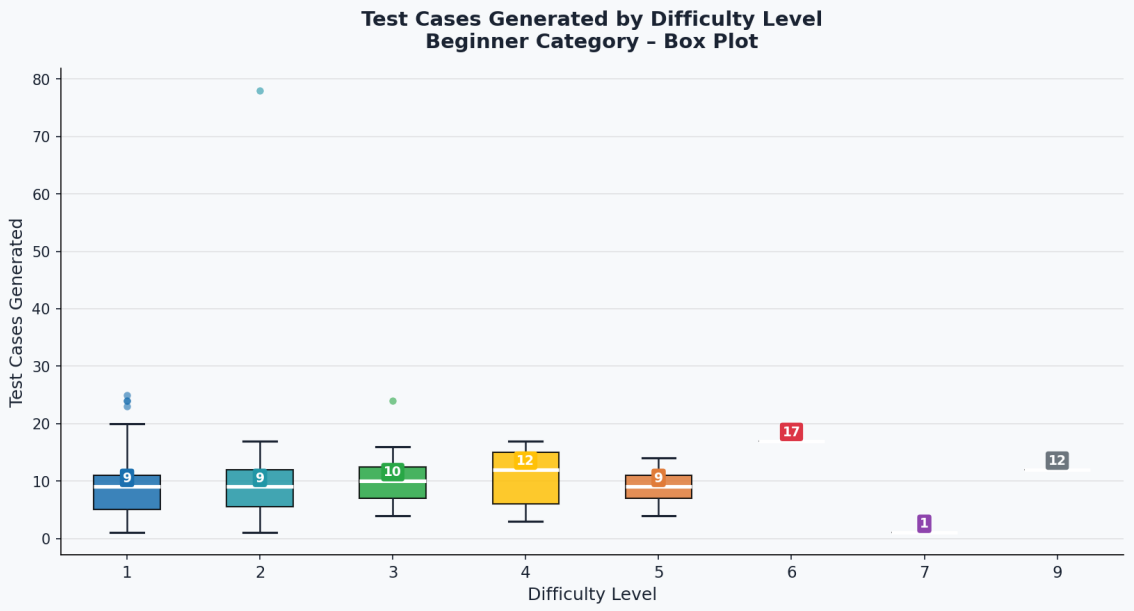


Figure 4.4: Test Case Generation Per Difficulty

the length of a test suite, two opposite cases will be examined, challenges 1155 and 1960.

4.3.2 Challenge 1155

Challenge 1155 illustrates a scenario in which only a single test case was generated. The task consists of computing the value of S, defined as $S = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{100}$. Given that the output is fully deterministic and can be derived directly from the specification, with no variation in input parameters, a single test case is sufficient to validate the correctness of any implementation.

4.3.3 Challenge 1960

In contrast, challenge 1960 requires implementing a conversion algorithm from Arabic numerals to Roman numerals (e.g., 999 $\rightarrow$ CMXCIX). The Roman numeral system is based on a fixed set of symbols (I, V, X, L, C, D, M) and includes subtractive notation rules such as IV for 4 and IX for 9, which introduce multiple structural edge cases. To ensure adequate coverage of these variations, the generative model produced a significantly larger test suite comprising 78 test cases, capturing a broad range of valid inputs and edge conditions.

4.3.4 Variability Conclusions

The results suggest that challenge difficulty is not the primary factor influencing the number of test cases generated. Although difficulty may have some impact, the nature and complexity of the problem specification appear to play a far more significant role in determining the size of the generated test suite.

This observation is supported by the considerable variability in the number of test cases across challenges of different difficulty levels. For example, the challenge that generated the most test cases was classified as Difficulty 2, whereas a Difficulty 7 challenge received only a single test case. As illustrated in Figure 4.4, no clear relationship can be established between challenge difficulty and the number of generated test cases.

Consequently, the findings indicate that the characteristics of the problem itself, such as the number of possible input scenarios, edge cases, and behavioural constraints, have a greater influence on test case generation than the platform's difficulty rating.

4.3.5 Test Execution Success Rate

It is essential to analyse the Test Execution Success Rate of the generated test suites for each challenge. In this context, a successful test execution is defined as a test case that, when executed, does not indicate any anomaly and complies with the intended behaviour. However, an observed anomaly may originate from different sources: it may correspond to an incorrect test case (i.e., one that does not accurately represent the intended behaviour of the algorithm), or it may reveal a genuine defect or uncovered edge case in the implementation. Distinguishing between these two cases would typically require manual inspection or an additional verification mechanism. Although one of the objectives of this study is to demonstrate that the workflow is autonomous, manual verification of each failing test case will be performed to identify which tests are detecting bugs and, for those that are not, the reasoning behind it.

Of the 207 analysed challenges, 58 (28.0%) contained at least one failing test case, with a maximum of 12 failures in a single challenge. Furthermore, 13 challenges exhibited a 0% success rate, indicating that all generated test cases failed. The remaining distribution is as follows: three challenges achieved a success rate below 26%, seven challenges fell below 51%, eleven challenges achieved a success rate between 51% and 75%, and 24 challenges achieved a success rate between 76% and 99% (see Figure 4.5).

A more detailed analysis suggests that the majority of observed failures are attributable to incorrect LLM-generated oracles rather than to failing test cases that are detecting defects in the underlying algorithm implementations. This indicates that the primary source of inconsistency lies in the test generation process itself, rather than in the solutions being evaluated.

Manual inspection of the generated test suites identified irregularities in 176 out of 1,954 test cases. After filtering these cases, the final dataset consists of 1,778 valid test cases, corresponding to 90.99% of the original set. Despite the availability of formal proofs as a validation signal, a manual inspection of all failing test cases was performed.

This inspection revealed that three challenges (1071, 1044, and 1036) contained implementation errors that were nevertheless accepted by the platform. For the remaining failing cases, two main categories of issues were identified:

1. **Incorrect Oracle Generation:** This category corresponds to test cases that do not correctly reflect the intended behaviour specified in the problem description, or that introduce inconsistencies unrelated to the algorithm under test. In such cases, the failure is caused solely by an incorrect

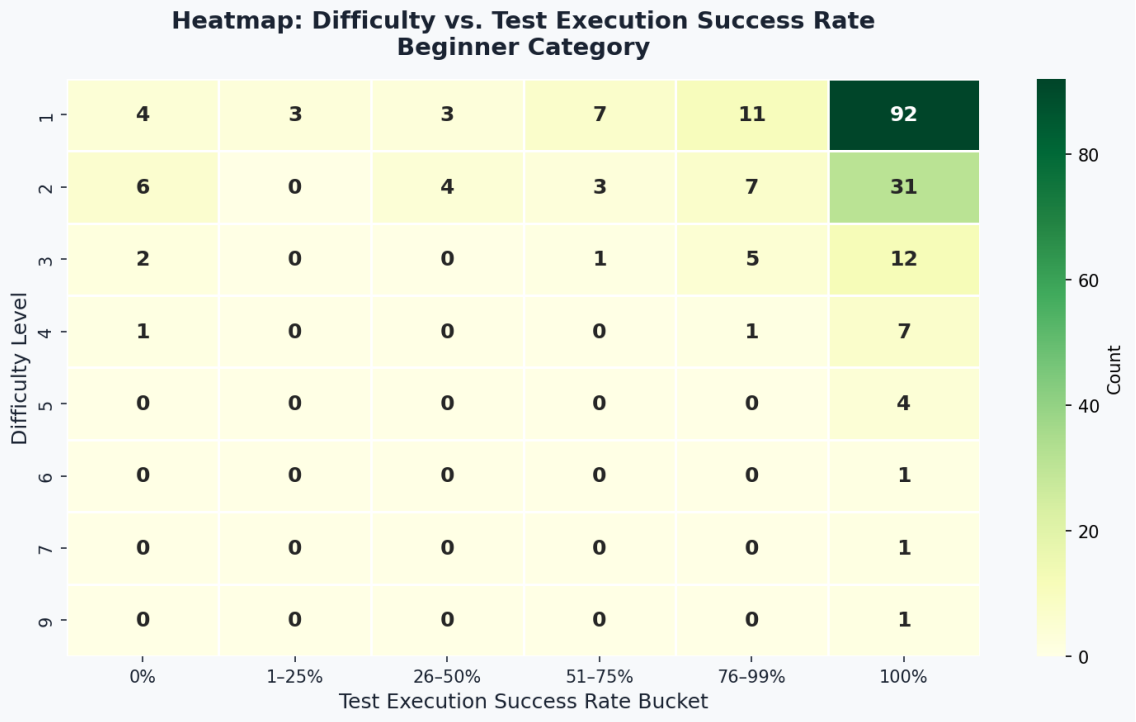


Figure 4.5: Test Execution Success Rate Heat Map

oracle rather than by a defect in the implementation. For example, a test case involving the division of two integers (e.g., 5 and 6) may incorrectly specify an expected output of 3, despite no valid justification for this result.

2. **Precision-Related Faults:** These errors occur when discrepancies arise due to floating-point precision limitations. In such cases, the expected output specified by the test case does not match the actual output due to insufficient numerical precision or rounding inconsistencies. A representative example of this issue is observed in Challenge 1116.

The issues identified under the category of **"test failure due to an incorrect oracle"** were primarily attributed to two causes: (1) compilation errors, and (2) cases in which the expected outputs were correctly specified in the problem description, but the generative AI tool failed to produce test cases in the required format or structure.

4.3.6 Test Failure Due To An Incorrect Oracle

An illustrative example of the first issue (compilation errors) can be observed in Challenge 1181. In this exercise, a 12×12 two-dimensional matrix is provided, and the objective is to compute either the sum or the average of selected floating-point values, depending on the problem specification.

In this case, the generative AI tool produced 12 test cases, none of which successfully passed validation. The failures were not solely due to incorrect expected outputs, but also because several generated test cases were not executable. The primary source of error was due to both inconsistencies in the expected data type (e.g., integers instead of floating-point values) and violations of the

required input structure. Specifically, some test cases contained more than the required 144 values, while others contained fewer, despite the implementation strictly requiring exactly 144 matrix entries. Overall, these results indicate a clear limitation of the generative process when handling structured matrix-based inputs, although it remains unclear whether this issue is primarily driven by ambiguities in the problem specification or by limitations of the generative AI tool itself.

A similar issue can be observed in another class of structured input problems. In this challenge, the program processes input lines consisting of three characters, each of which is either $\star$ or $-$. The symbol $\star$ is interpreted as the binary value 1, while $-$ is interpreted as 0, forming a 3-bit binary representation. For example, the pattern $\star--$ corresponds to the decimal value 4. These values are accumulated until the sentinel input `caw caw` is encountered, at which point the current sum is stored, and the accumulator is reset. After three such sums are computed, the program outputs each result on a separate line.

In this case, the generative AI tool failed to consistently preserve the required input structure in 2 out of 24 generated test cases, most notably by omitting the terminating `caw caw` line. Similar issues with input format consistency were also observed in other matrix-oriented and structured parsing problems, suggesting a recurring limitation in handling strict input protocols.

Challenge 1848

Input Sample	**Output Sample**
`*−− caw caw`	4

`*−− caw caw`
`caw caw`
`caw caw`
	0

4.3.7 Precision Fault

Regarding **precision-related faults**, Challenge 1079 provides a representative example. This exercise requires computing a weighted average of three numerical values (x), (y), and (z) using weights of 20%, 30%, and 50%, respectively.

In this case, two test cases fail not due to logical or implementation errors but due to discrepancies in numerical precision. The exact input–output pairs generated by the GenAI tool are presented below.

A manual verification indicates that the discrepancy arises from intermediate weighted values such as (-1.25) and (2.25). Although the implementation is expected to adhere to the precision rules defined in the problem statement, which typically involve a fixed number of decimal places, the GenAI-generated expected outputs appear to use an alternative rounding strategy. This inconsistency likely stems from ambiguity in how numerical precision should be handled, particularly

regarding the number of decimal places retained during intermediate computations versus those retained in the final output.

Challenge 1079

Input Sample	Output Sample
1 -0.5 0.0 -0.5	-1.3
1 11.0 0.0 0.1	2.3

4.3.8 Faults Detected

In terms of **confirmed implementation faults**, four challenges were identified, with three being challenges with examples of test cases that proves that the implementation does not comply with the specification and one challenge with an unclear specification resulting in a curious case. These challenges are:

- **Challenge 1036**: Compute the roots of Bhaskara's formula and return the real roots.
- **Challenge 1044**: Determine whether two integers are multiples of each other.
- **Challenge 1071**: Compute the sum of all odd numbers between two given integers.
- **Challenge 1116**: Perform division and print the result with a precision of one decimal place.

In the following subsections, each case is analysed in detail, focusing on the nature of the detected defects and their implications for the dataset's overall reliability.

4.3.8.1 Challenge 1036

The test case that exposes this anomaly is the input (-1, 2, -1), which yields a single real root. The issue arises because the implementation does not correctly handle the case where $b^2 - 4ac = 0$, which should yield a single repeated real root rather than an exceptional or undefined result. Instead, the algorithm fails to account for this boundary condition, leading to incorrect computation and an "Impossible to calculate" output.

Challenge 1036

Input Sample	**Output Sample**
1 -3 2	R1 = 2.00 R2 = 1.00
2 4 2	R1 = -1.00 R2 = -1.00

4.3.8.2 Challenge 1044

The platform solution omits an explicit condition ensuring that no input results in a zero divisor. Although this represents a well known edge case in modular arithmetic and divisibility checks, it is not clearly specified in the problem description. Consequently, the GenAI tool generated test cases involving zero valued divisors, which in turn exposed this ambiguity in the specification.

This observation suggests that such cases require a more precise definition within the problem statement, either by explicitly defining the expected behaviour for zero divisors or by ensuring they are excluded from the input domain. In particular, these cases should be unambiguously mapped to a specific output category, such as “They are not multiples”, or alternatively handled through a clearly stated exception rule.

In a learning environment such cases must not occur since it will hinder the learning process.

Challenge 1044

Input Sample	**Output Sample**
-5 12	They are not multiples
10 5	They are multiples

4.3.8.3 Challenge 1071

The test case revealing this issue is the input (1, 10), which should yield a result of 24 rather than 16. The discrepancy arises from an incorrect loop implementation in the algorithm, which fails to include certain values that should contribute to the final sum. As a consequence, the computed output is incomplete, reflecting a systematic omission rather than an arithmetic error.

Challenge 1071

Input Sample	Output Sample
1 10	24
6 -5	5

4.3.8.4 Challenge 1116

The final anomaly concerns Challenge 1116, which is also classified as a **precision-related fault**. In this case, the underlying implementation logic is correct; however, the output formatting does not adhere to the required numerical precision specified in the problem statement.

For example, given the input ((34, 100)), the expected output is (0.3), whereas the implementation produces (0.34). This discrepancy is not due to an error in the computation itself, but rather stems from an incorrect handling of output formatting, specifically the number of decimal places required in the final result.

Challenge 1116

Input Sample	Output Sample
34 100	0.3
-7 2	3.5

4.3.9 Discussion of results

Overall, the results obtained in this phase are promising. Out of the 207 analysed challenges, only 58 (28%) exhibited anomalies within their generated test suites. When focusing exclusively on these problematic cases, the average Test Execution Success Rate is 54.1%. These results suggest that, while the proposed approach is capable of generating valid and useful test cases in many situations, the performance of the LLMs could still be improved.

As previously discussed, 13 challenges achieved a Test Execution Success Rate of 0%. Notably, the majority of these cases (9 out of 13) correspond to matrix-based problems. These challenges were therefore identified as a significant obstacle within the proposed workflow. The underlying cause of this behaviour remains unclear. It is possible that matrix-oriented specifications introduce additional complexity for the LLM, particularly with respect to generating structured inputs that strictly conform to the required dimensions and formatting constraints. However, further investigation is required before drawing definitive conclusions.

The subsequent phases seek to determine whether this limitation can be mitigated through prompt refinement and additional verification mechanisms. In particular, they assess whether the

difficulties observed in matrix-based challenges persist or can be substantially reduced through iterative improvements to the generation process. The distribution of Test Execution Success Rate (TESR) obtained with the original prompt is illustrated in Figure 4.8 on the left side (data on red). Figure 4.8 (left side) in the X axis the code of all the challenges that have a TESR below 100%, and on the Y axis the exact percentage TESR of each challenge. It is possible to verify the identified 13 challenges with a TESR of 0%.

With manual inspection, there were three challenges that presented anomalies with its algorithm, challenges 1071, 1036 and 1116. Additionally, challenge 1044 could be considered to accept an incorrect implementation, but due to an ambiguity in the specification, it will not be considered.

4.3.10 Answering RQ2

The results obtained in this phase provide the necessary evidence to address Research Question 2. The objective of this research question is to evaluate whether LLM-generated test cases can support the assessment of the feedback provided by programming learning platforms, particularly by identifying inconsistencies in the acceptance of submitted solutions.

- **RQ2: Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of test cases?**

  Answer: The findings indicate that LLM-generated test cases can effectively assist in the automated evaluation of programming learning platforms. Despite some invalid test cases and oracle inaccuracies, the generated test suites identified accepted solutions that did not fully comply with their corresponding specifications. More specifically, within the *Beginner* category, i.e, where accepted solutions would generally be expected to be correct, the generated test cases revealed at least three challenges containing implementations that should not have been accepted by the platform.

4.4 Phase 3: Conditions and Formal Proof Generation

After generating test cases, it is still necessary to define preconditions and postconditions and construct the corresponding formal proofs to establish a verification mechanism that can signal potential anomalies in the implementations.

The generation of pre- and post-conditions proceeded without issues, and all 207 challenges were successfully processed. However, the formal proof generation phase revealed several notable irregularities.

In the vast majority of cases, no inconsistencies were identified. More precisely, only 3 out of 207 exercises exhibited either detected anomalies or discrepancies when compared with the results obtained through manual verification in the previous phase. Specifically, challenges 1036 and 1071 were classified as incorrect, while Challenge 1044 was classified as partially correct. In addition, Challenge 1181 required regeneration due to an invalid or inconsistent proof (see Figure 4.6). A

further exceptional case was Challenge 1116, which was classified as totally correct despite having been previously identified as problematic during the test execution phase.

In the following sections, each of these cases is analysed in detail.

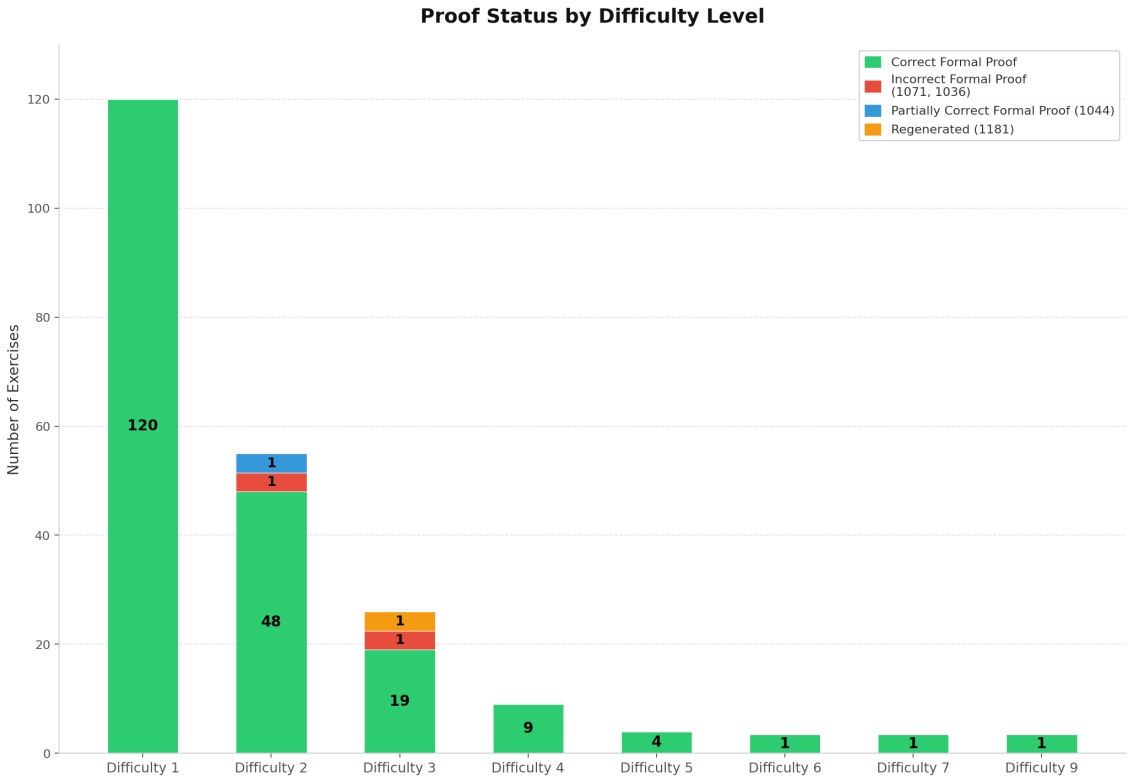


Figure 4.6: Generated Proofs By Difficulty

4.4.1 Partially Correct Proofs

As previously mentioned, Challenge 1044 was classified as *Partially Correct*. This challenge requires determining whether two integers are multiples of each other (see 4.3.8.2).

Although a missing edge case involving zero divisors had already been identified in the previous phase, the generated formal proof still classified the implementation as partially correct. The proof further states that "*if we also add the simple guard that prevents a modulo-by-zero, the program terminates for all integer inputs, giving us total correctness*".

This behaviour is particularly noteworthy, as the LLM correctly recognised the absence of a critical edge case from both the specification and the derived pre- and postconditions, yet did not classify the implementation as incorrect. Instead, it assigned a partial correctness label. Although this was the only occurrence of such behaviour across the entire experimental set, it raises questions regarding whether the outcome is influenced by ambiguity in the problem specification or by limitations in the reasoning strategy of the model. Further analysis is required before drawing definitive conclusions.

4.4.2 Incorrect Proofs

As previously mentioned, Challenges 1036 and 1071 were identified as incorrect (see 4.3.8.1 and 4.3.8.3). In both cases, the LLM concluded that the proposed implementations do not fully comply with the given specifications and, notably, identified the same issues that had already been observed during Phase 2.

For Challenge 1036, the generated proof states that "*the program is partially correct with respect to the given specification.*". However, it further adds that "*If we also add the simple guard that prevents a modulo-by-zero, the program terminates for all integer inputs, giving us total correctness.*". This indicates that the correctness of the implementation is conditional on an additional safeguard that is not present in the original solution. As such, the algorithm cannot be considered correct in its current form, since it fails to satisfy the specification without modification.

For Challenge 1071, the proof explicitly states that *"Both loops skip one endpoint and keep the other"*, and consequently concludes that *"Consequently they do not satisfy the specification"*. This conclusion directly matches the issue previously identified in the test execution phase, confirming that the implementation systematically omits required boundary values and therefore deviates from the expected behaviour defined in the specification.

4.4.3 Regenerated Proof

Another relevant case to mention, which is challenge 1181. Initially the LLM suggested that the implementation did not comply with the problem specification, this however was disproved after a manual inspection of the proof. The initial formalization done by the LLM of the problem was not the correct, and therefore this entire phase was redone for this specific challenge.

In a second attempt the LLM correctly formalized the specification and was able to detect that the implementation for this challenge is Totally Correct. Lets brake it down.

The task consists of computing either the sum or the arithmetic mean of a selected set of matrix elements, identified by the row index L $0 \leq L \leq 11$. According to the official specification, the relevant set includes all elements located in row (L), namely

$$\{M[L][j] \mid 0 \leq j \leq 11\}.$$

The required operation is determined by the input parameter $T \in \{\text{'S'}, \text{'M'}\}$, which specifies whether the result should be the sum or the average of the selected elements.

Since executable test cases were not available during the previous stages, this phase plays a critical role in assessing whether the implementation satisfies the intended specification.

The generated formal proof initially indicates that the implementation is incorrect. According to the proof, the algorithm fails to process the appropriate set of elements and suggests an alternative correction based on a different interpretation of the problem. More specifically, the proof assumes that the computation should be performed over the submatrix:

$$\{M[i][j] \mid 0 \leq i \leq L \land 0 \leq j \leq L\},$$

Representing the upper-left $(L+1) \times (L+1)$ portion of the matrix.

However, this interpretation is inconsistent with the original problem statement. The specification explicitly states that the computation must be carried out using the elements of a single row rather than a two-dimensional matrix region. Consequently, the observed discrepancy results from an incorrect formalization of the problem requirements rather than from a defect in the implementation itself.

As said previously, after this manual inspection it was needed to generate again the formal proof in to comprehend whether the same misunderstanding would reoccur. Although the problem description remained unchanged, the preconditions and postconditions were also newly generated. The resulting formal proof demonstrates that the implementation conforms to the specification and is therefore correct.

4.4.4 Challenge 1116 Proof

Finally, Challenge 1116 yielded an unexpected result. As previously discussed, this challenge includes a specific requirement regarding the formatting of the output (see 4.3.8.4). Moreover, there exists a test case capable of revealing that the retrieved solution does not fully satisfy the problem specification. Although the implementation is logically correct and computes the expected values, its output formatting does not strictly adhere to the required precision constraints, and it should therefore be considered incorrect with respect to the specification.

Given this discrepancy, it was anticipated that the formal proof would similarly classify the solution as problematic. However, this was not the outcome. Instead, the GenAI tool justified its conclusion by arguing that the formatting precision requirement was not the primary focus of the challenge. On this basis, it classified the solution as entirely correct, despite the observed deviation in output formatting.

4.4.5 Discussing Results

This phase aimed to evaluate whether formal proofs can be used as an additional mechanism to identify defective implementations. A total of 207 formal proofs were generated. Of these, 204 classified their corresponding solutions as *totally correct*, 2 as *incorrect*, and 1 as *partially correct*.

Overall, the formal verification process successfully identified most of the problematic implementations detected in the previous phase. More specifically, three of the four challenges previously identified as containing defects were also flagged by the generated formal proofs. The main exception was Challenge 1116, which produced an unexpected result. Although the formal proof explicitly acknowledged the discrepancy in the output formatting requirements, it concluded that this issue was not sufficiently significant to invalidate the implementation. Consequently, the solution was classified as *totally correct*, despite failing to comply with a constraint explicitly stated in the problem specification.

Another noteworthy case was Challenge 1044, which was classified as *partially correct*. Similar to the findings from the previous phase, the generated proof identified an ambiguity in the

handling of zero divisors. However, rather than considering this omission sufficient to invalidate the implementation, the proof concluded that the solution remained partially correct. This result suggests that, in some situations, the LLM may recognise a defect while simultaneously underestimating its impact on the implementation's overall correctness.

An additional irregularity was observed in Challenge 1181. The initial formal proof incorrectly classified the implementation as defective due to an inaccurate formalisation of the problem specification. Following a manual inspection, the proof generation process was repeated, resulting in a correct formalisation and a final classification of the implementation as *totally correct*. This case highlights that inaccuracies in the formalisation stage may propagate to the proof itself, potentially leading to incorrect conclusions.

Despite these isolated cases, the overall results demonstrate strong alignment between the outcomes of the formal verification process and those from specification-based test generation. In most instances, the generated proofs identified the same implementation inconsistencies that had previously been detected by failing test cases. The primary differences arose from how the LLM interpreted the severity of those inconsistencies rather than from its ability to detect them. Consequently, the findings suggest that formal proofs can serve as an effective complement to identifying potentially defective implementations and strengthening confidence in the validation process.

4.4.6 Answering RQ3

The results obtained in this phase provide the necessary evidence to address Research Question 3. The objective of this research question is to evaluate whether LLM-generated formal proofs can support the assessment of the feedback provided by programming learning platforms, particularly by identifying inconsistencies in the acceptance of submitted solutions.

- **RQ3: Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of formal proofs?**

  Answer: The findings indicate that LLM-generated formal proofs can assist in the automated evaluation of programming learning platforms. The generated proofs successfully identified most of the problematic implementations previously detected through specification based test generation, thereby reinforcing the consistency of the results obtained across both validation mechanisms.

  Although a small number of unexpected outcomes were observed, the formal proofs generally recognised the same discrepancies between the implementations and their corresponding specifications. In most cases, the identified irregularities were correctly highlighted, even when the final classification assigned by the LLM did not fully reflect the severity of the detected issue. Furthermore, only one proof required regeneration due to an incorrect formalisation of the problem specification, suggesting that the overall proof generation process was highly reliable.

These results demonstrate that LLM-generated formal proofs can serve as an effective complementary validation mechanism. When combined with specification-based test generation, they provide an additional layer of confidence in the assessment process and help identify accepted solutions that may not fully comply with the intended specifications.

4.5 Phase 4: Initial Generalisation Evaluation

In this Phase it will be verified an initial indication of generalizability of entire workflow. We will extend the initial 207 challenges by an additional 12, chosen randomly. Three challenges were chosen from four different categories, namely, “Strings”, “Mathematics”, “Data Structure & Libraries", and “Ad-hoc”. The gathered challenges followed the exact same methodology for the solution and specification, with a little nuance, since there was a lack of Python solutions. The Java version was used as the basis, and with Claude Sonnet 4.6, a translation to Python was produced. Every solution was verified and accepted by the platform before investigating further.

4.5.1 Challenges Difficulty

Once again, it is important to examine the distribution of difficulty levels within the selected set of challenges. The 12 challenges analysed in this phase span difficulty levels ranging from 3 to 10, with an average difficulty of 5. The challenge with the lowest difficulty rating is Challenge 1367 in the “Strings” category, while the most difficult challenge is Challenge 3105, assigned a difficulty level of 10.

The distribution of challenge difficulties is illustrated in Table 4.2.

This broader range of difficulty levels provides an opportunity to further investigate whether challenge difficulty has a measurable impact on the quality of test case generation. In particular, it will be valuable to assess whether more difficult problems lead to lower test execution success rates, a greater number of generated test cases, or an increased frequency of oracle-related errors compared to less complex challenges. Such an analysis may help determine whether the difficulty of the challenge is a significant factor influencing the effectiveness of the proposed methodology.

4.5.2 Generated Test Cases & Test Execution Rate

Focusing now on the generated test suites, a total of 132 test cases were produced across the 12 analysed challenges, averaging 11 per challenge. The largest test suites were generated for Challenges 1161 and 1026, each containing 20 test cases, while Challenge 2018 received the smallest test suite with 5 test cases. The complete distribution of generated test cases is presented in Table 4.3.

As observed in the previous experiment, no clear relationship exists between challenge difficulty and the number of test cases generated. The distribution does not reveal any discernible pattern, indicating that more difficult challenges systematically result in larger or smaller test suites.

Table 4.2: Difficulty Levels of Challenges in Generalisation Phase

Category	Challenge ID	Difficulty Level
Strings	#2062	4
	#1367	3
	#1898	9
Mathematics	#2890	5
	#1492	5
	#1161	5
Data Structures & Libraries	#1069	5
	#2018	4
	#1281	3
Ad-HOC	#1026	5
	#1495	6
	#3105	10

Consequently, based on the analysed sample, challenge difficulty alone does not appear to be a significant factor affecting the test case generation process.

Table 4.3: Test Cases Generation in Generalisation Phase

Category	Challenge ID	Test Cases #
Strings	#2062	10
	#1367	6
	#1898	7
Mathematics	#2890	13
	#1492	18
	#1161	20
Data Structures & Libraries	#1069	6
	#2018	5
	#1281	7
Ad-hoc	#1026	20
	#1495	10
	#3105	10

Despite the substantial number of test cases generated, it is still necessary to evaluate whether

they are suitable for effective validation. After their generation, all test cases were executed against their corresponding implementations using the same procedure as for the main dataset.

The results were particularly encouraging: all 12 challenges achieved a Test Execution Success Rate of 100%. In other words, every generated test case was successfully executed and produced outputs consistent with those expected by the reference implementations.

This result represents an important milestone in the evaluation of the proposed methodology. It provides preliminary evidence that the workflow is not limited to the "Iniciante" category and may be applicable across different problem domains and difficulty levels. The absence of execution failures suggests that the generated test suites correctly captured the expected input formats and behavioural requirements of the selected challenges, regardless of category.

Nevertheless, these findings alone are insufficient to establish the approach's generalizability. While a perfect Test Execution Success Rate indicates that the generated test cases are executable and internally consistent, it does not guarantee that the underlying implementations are correct or that the generated oracles are free from defects. For this reason, it is necessary to examine the results of the formal verification phase before drawing stronger conclusions regarding the effectiveness and general applicability of the proposed workflow.

4.5.3 Conditions & Formal Proof Generation

The final step of this phase involved generating the preconditions and postconditions for each challenge, followed by constructing the corresponding formal proofs. As in the base experiment, no difficulties were encountered in generating the conditions. The primary objective at this stage was therefore to determine whether the formal verification process would identify any anomalies within the selected implementations.

Interestingly, the formal proofs did not detect any anomalies among the 12 analysed challenges. All implementations were classified by the AI tool as *totally correct*, indicating full compliance with their respective formalised specifications.

4.5.4 Phase 4 Conclusions

These results are particularly noteworthy when considered alongside the findings from the previous phase. Not only did all generated test cases achieve a 100% Test Execution Success Rate, but the subsequent formal verification process also failed to identify any inconsistencies or specification violations. Taken together, these findings provide additional evidence supporting the applicability of the proposed methodology beyond the initial "Iniciante" category.

Nevertheless, caution is still warranted when interpreting these results. As demonstrated in the main experiment, formal proofs generated by LLMs are not immune to errors and may occasionally misrepresent the original problem specification. Consequently, the absence of detected anomalies should not be interpreted as definitive proof that all implementations are correct. Rather, it indicates that the proposed verification workflow identified no inconsistencies for this particular set of challenges.

Overall, the results of this phase suggest that the methodology exhibits a promising degree of generalizability across different categories and difficulty levels. However, a larger and more diverse sample would be required to draw stronger conclusions regarding its effectiveness in broader contexts.

4.6 Phase 5: Prompt Refinement Evaluation

Overall, the experiment has yielded solid results. In the initial evaluation of over 207 challenges, only 57 exhibited anomalies in the generated test cases, while the remaining cases executed successfully and were consistent. Subsequently, applying the same workflow in a broader context also yielded promising outcomes and provided preliminary evidence for the generalisability of the proposed methodology, although this claim requires further validation on a larger, more diverse set of challenges than the 12 considered in the generalisation study.

A particularly interesting property of large language models is their capacity to adapt and improve when provided with explicit feedback on prior errors. In this sense, GenAI-based tools can iteratively refine their outputs when guided by appropriately structured corrective information. In this phase, this capability is explicitly examined.

To this end, the initial prompt is refined using the subset of exercises that exhibited the most significant issues, specifically those related to matrix-based problems. In these cases, the original problem descriptions, the generated test cases, and the initial prompt are provided as input to *Claude Sonnet 4.6* to derive an improved prompt. This refinement process aims to address the observed shortcomings and improve the previously recorded Test Execution Success Rate of 0% across all challenges in this category.

The resulting output of this refinement process is the prompt presented in Figure 4.9. The primary modification between the original and refined versions lies in the increased emphasis on input and output formatting constraints, which were identified as a key source of failure in earlier phases of the study. It is important to clarify that, in this phase, the only “artefacts” to be regenerated are test cases.

4.6.1 Results

Exploring the outcomes of this final experiment reveals several notable findings. After applying the refined prompt, an overall improvement in the Test Execution Success Rate was observed. Considering the 57 exercises that initially exhibited anomalies in their test suites during the first phase, the baseline execution success rate was 54.86%. While this value is not critically low, it nevertheless indicates a substantial presence of issues affecting test reliability and consistency.

Following the introduction of the refined prompt, the execution success rate increased significantly, reaching 70.9% (see Figure 4.7). In addition, 12 exercises achieved a 100% Test Execution Success Rate, the majority of which corresponded to cases that previously exhibited a 0% success rate (See Figure 4.9). Only challenge 1190 from the matrix-themed challenges remained with a Test Execution Rate of 0%. This indicates that the prompt refinement was particularly effective in

addressing the most problematic cases identified in the initial phase. The average Test Execution Success Rate is 74,1% as shown in Figure 4.8 on the right. The aforementioned Figure shows a clear improvement over the initial results obtained with the base prompt, with more cases of challenges at a TESR of 100% and only one challenge, 1190, that remained unchanged.

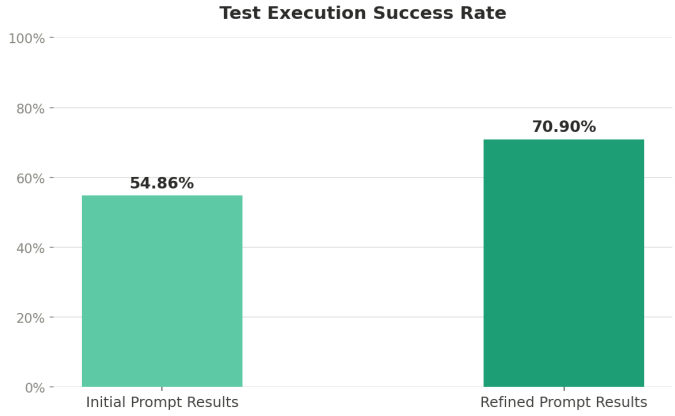


Figure 4.7: Test Execution Success Rate Comparison

This also highlights that prompt optimisation does not necessarily produce uniform improvements across all cases since some challenges appear a TESR very similar as the initial prompt.

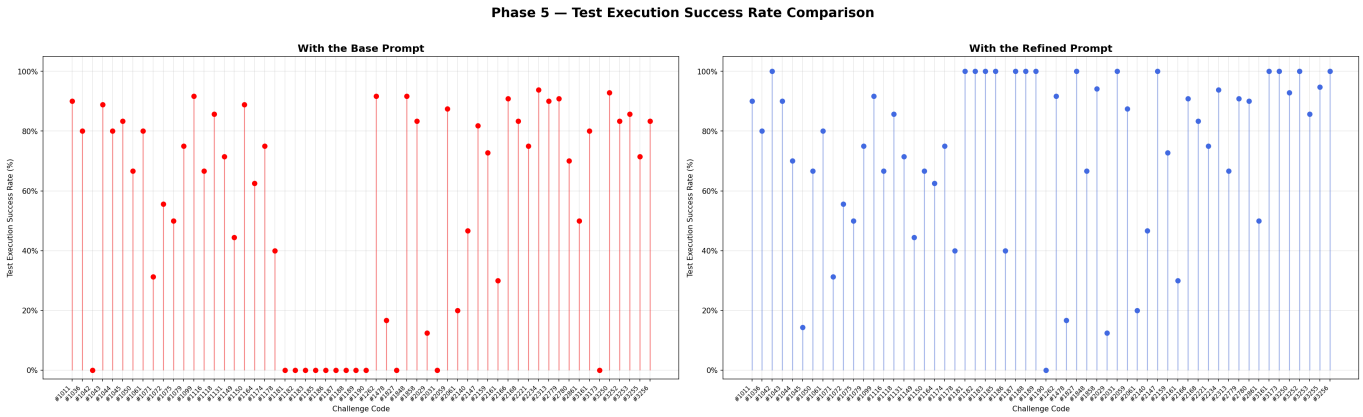


Figure 4.8: Test Execution Success Prompts Comparison

Overall, these findings are highly encouraging and provide a direct answer to the final research question, namely: “To what extent can prompt refinement improve the effectiveness of specification-based test oracle generation?” The results indicate that meaningful improvements are achievable by incorporating feedback from previously observed failures into the LLM prompt, demonstrating that iterative refinement can substantially enhance performance, although this is not guaranteed in every case, as shown in Figure 4.8.

4.6.2 Answering RQ4

The results obtained in this phase provide the necessary evidence to address Research Question 4. The objective of this research question is to evaluate the extent to which prompt design influences the quality and effectiveness of LLM-generated test cases.

- **RQ4: Could the prompt structure affect the results regarding the test cases generated by the LLM?**

  Answer: The findings clearly indicate that prompt structure has a significant impact on the quality of the generated test cases. Refining the initial prompt based on the difficulties and failure patterns identified during Phase 2 resulted in a substantial improvement in performance. More specifically, the overall Test Execution Success Rate of the challenges that previously contained failing or non-executable test cases increased by over 15 percentage points, rising from 54.86% to 70.9%.

  Furthermore, the refined prompt significantly reduced the number of challenges with a 0% Test Execution Success Rate, while increasing the number of challenges for which all generated test cases executed successfully. These results demonstrate that prompt design plays a crucial role in specification-based test generation and that incorporating knowledge gained from previous failures can substantially improve the reliability and executability of LLM-generated test suites.

  Consequently, the evidence suggests that prompt refinement is an effective means of improving test-generation outcomes and can be viewed as a practical form of iterative enhancement within AI-assisted software testing workflows.

4.7 Answering Research Question 1

The primary research question guiding this dissertation is whether feedback from online programming learning platforms can be considered reliable. The previous sections addressed the supporting research questions and examined the effectiveness of test generation, formal verification, and prompt refinement in identifying inconsistencies within the platform's assessment process. Collectively, these findings provide the necessary evidence to answer the overarching research question.

- **RQ1: Can we trust the feedback provided by programming learning platforms?**

  Answer: The results suggest that the feedback provided by programming learning platforms should not be assumed to be entirely reliable. Although the vast majority of accepted solutions analysed in this study were correct, the proposed methodology identified multiple cases in which Platform X accepted implementations that did not fully comply with their corresponding specifications. These inconsistencies were detected through both specification based test generation and LLM-generated formal proofs, which revealed overlooked

edge cases and specification violations that were not captured by the platform's existing validation mechanisms.

While the number of defective accepted solutions was relatively small, their existence demonstrates that the platform's feedback system is not infallible. This observation is particularly relevant given that the identified anomalies were found within the *Beginner* category, which is specifically intended for novice programmers. Since learners often rely on the platform's acceptance decision as an indicator of correctness, the presence of accepted yet defective solutions may negatively affect the learning process and reinforce incorrect programming practices.

Consequently, the findings indicate that programming learning platforms can generally be trusted to provide useful feedback, but their assessments should not be considered definitive. Additional validation mechanisms, such as specification-based test generation and formal verification techniques, can serve as valuable complementary approaches for increasing confidence in the correctness of accepted solutions and identifying cases where the platform's evaluation process may be insufficient.

Refined Prompt

```
You are acting as a strict black-box software tester.
You will receive a programming problem description.
Your task is to generate a comprehensive and non-redundant set of
test cases that fully exercise the problem specification.
BEFORE generating any test case, you must:

  1. Carefully parse the input specification:  identify the exact
     number of lines, the type and format of each line, any ordering
     constraints, and any dependencies between inputs.
  2. Carefully parse the output specification:  identify the
     exact format, precision, number of lines, and any conditional
     formatting rules.
  3. Identify ALL constraints (minimum, maximum, boundary values)
     for every input field.
  4. List the distinct behavioural paths the problem has (e.g.,
     different operation modes, edge values, sign changes).
     Only then generate test cases that satisfy all of the above.

Requirements:

    • Cover:  normal cases, edge cases, boundary conditions,
      minimum/maximum constraints, and all behavioural paths implied
      by the specification.
    • Do NOT reuse, restate, or modify any example given in the
      problem.
    • Do NOT generate redundant or equivalent test cases.
    • Stop generating cases when additional cases would be redundant.

Input construction rules:

    • Every input must strictly and completely conform to the input
      specification.
    • Respect the exact number of values, lines, and tokens required.
    • Respect all data types (integer, float, character, string)
      exactly as specified.
    • Respect all value ranges and constraints for every field.
    • If the input requires N values, generate exactly N values --
      never fewer, never more.
    • If values are positional (e.g., a matrix filled row by row),
      place values in the correct positions.

Output construction rules:

    • Derive the expected output by mentally simulating the correct
      algorithm on your input.
    • Respect the exact output format:  number of lines, separators,
      decimal precision, spacing.
    • If output precision is specified (e.g., one decimal place),
      apply it exactly.
    • If the output is conditional on the input (e.g., different
      operations), verify the condition and apply the correct
      formula.
```

Figure 4.9: Refined prompt to support test generation

Chapter 5

Conclusions & Future Work

Investigating if we can trust the feedback provided by programming learning platforms has proven to be both an insightful and rewarding research endeavour. Throughout the design, implementation, and evaluation of the proposed framework, each phase introduced distinct challenges that demanded careful analysis and critical reflection. The obstacles encountered, together with the lessons learned and insights gained, represent valuable contributions that may guide future research efforts, helping other researchers avoid similar difficulties while building upon the findings presented in this work. Furthermore, the study uncovered several opportunities for advancing the state of the art in specification based test generation, LLM generated formal verification and in the context of online programming platforms.

This dissertation has demonstrated that we cannot trust blindly in the feedback from online programming languages, and there is a framework that can assist in this subject. Although the results reveal important minor limitations, particularly regarding oracle correctness, formatting constraints, and the handling of complex input structures, they also highlight the significant potential of LLM-based approaches when combined into a unified framework.

To conclude this dissertation, the principal findings of the research are revisited, providing a concise synthesis of the analyses conducted throughout the study and highlighting the key insights, challenges, and outcomes associated with each experimental phase. The research contributions are subsequently discussed in relation to the identified research gap, demonstrating how the proposed work advances current knowledge in this domain. Each research question is then revisited and answered based on the empirical evidence gathered during the experiments. Finally, several directions for future work are presented, including potential extensions of the proposed methodology and additional research opportunities that emerged from the results of this study, thereby laying the groundwork for further investigation into the role of Generative AI in software testing and verification.

5.1 Research Findings

Regarding the findings of this study, Phase 1 was the only stage that did not reveal significant insights directly related to the research questions. Its primary objective was to retrieve Python solutions and verify their acceptance by Platform X. Nevertheless, an important practical lesson emerged from this phase: studies involving online judge platforms require a high degree of automation. Verifying the correctness of 207 collected solutions proved to be a time consuming process, and three challenges required manual correction before being accepted by the platform. When scaled to larger datasets, such a task would quickly become impractical without automated support mechanisms. Consequently, automation should be considered a fundamental requirement for future studies of a similar nature.

Phase 2 proved to be one of the most insightful stages of the dissertation, largely due to the manual inspection performed on all failing test cases. Before discussing the findings themselves, it is important to highlight that the selection of the AI model is a critical decision, as it can significantly influence both the quality of the generated artifacts and the overall cost of the experimentation process.

From the 1,954 generated test cases, only 176 were classified as invalid, either because they failed during execution or because they could not be executed at all. This corresponds to approximately 9% of the generated test suite, indicating that the vast majority of generated tests were valid. At the challenge level, 58 out of the 207 analysed exercises contained at least one anomalous test case, representing 28% of the entire dataset. Among these anomalies, the two predominant categories were incorrect oracle generation and precision-related faults.

A particularly noteworthy finding was the existence of a subset of matrix-oriented challenges that consistently presented difficulties during test generation. Among the 13 challenges that achieved a Test Execution Success Rate of 0%, six were matrix-related problems, suggesting a systematic limitation of the approach when dealing with highly structured input formats. Although the root cause of this behaviour could not be conclusively determined, the consistency of the observed failures indicates that matrix-based problems represent a challenging class of specifications for current LLM based test generation techniques.

Furthermore, the manual analysis of the failing test cases revealed four problematic challenges, however it can be confidently said that three of them accepted solutions that did not respect their entire specification. This finding demonstrates that the generated test suites were capable of identifying genuine defects in accepted implementations. Although the number of detected defects was relatively small, it provides evidence that specification based test generation can contribute to defect discovery. The results of Phase 2 suggest that LLM generated test can help to evaluate the feedback of online academic platforms. In a category dedicated to newcomers into programming it was expected to find that every challenge would have a robust enough feedback system to ensure that no incorrect solutions was accepted, this however would be disproven by the LLM generated test cases.

Phase 3 provided additional valuable insights through the generation of preconditions, postconditions, and formal proofs. The generation of preconditions and postconditions proceeded smoothly, with no cases requiring regeneration or substantial correction. This suggests that LLMs are somewhat capable at extracting formal constraints from natural language specifications.

The formal proof generation process also demonstrated a high degree of reliability. The vast majority of generated proofs were consistent with the findings obtained through manual inspection in the previous phase. Only two challenges exhibited unexpected behaviour. In one case, the initial proof incorrectly classified a correct implementation as faulty due to an erroneous formalisation of the specification. In the other, the proof classified an implementation as totally correct despite the omission of an output formatting requirement explicitly stated in the problem description. Despite these exceptions, the results indicate a remarkable level of consistency. Since only 2 out of 207 proofs exhibited noteworthy discrepancies, it can be argued that the formal verification phase achieved approximately 99% agreement with the manually validated results. Moreover, all previously identified implementation anomalies were successfully detected by the generated proofs, reinforcing confidence in the usefulness of formal verification as an anomaly detection mechanism. Overall, this phase proved that it could also assist in evaluating the feedback of online academic platforms since it was able to identify that the platform accepted solutions that did not comply with the specifications.

Phase 4 provided evidence supporting an initial indication of the proposed methodology's generalizability. By applying the workflow to 12 challenges from different categories and difficulty levels, it was possible to evaluate whether the observed results were specific to the "Iniciante" category or reflected a broader applicability. Neither the test generation phase nor the formal verification phase encountered notable difficulties. Furthermore, all 12 solutions were classified as totally correct by the generated formal proofs, and no contradictory evidence emerged from the generated test suites. Although the sample size remains limited, these findings suggest that the methodology is not restricted to a single category of programming problems and may be applicable across a wider range of contexts.

Finally, Phase 5 demonstrated that previous failures can be effectively leveraged to improve future performance. Inspired by self-repair and iterative refinement mechanisms discussed in the literature, this phase explored whether the prompt itself could be improved based on the obstacles identified in earlier stages and, more objectively, whether the prompt structure could impact results in the test generation phase. The results were highly encouraging. The number of challenges with a Test Execution Success Rate of 0% decreased from 13 to only 1. Additionally, the overall Test Execution Success Rate increased from 54.86% to 70.9%, while the average Test Execution Success Rate within the problematic subset increased from 54.1% to 74.1%. These findings demonstrate that prompt refinement constitutes a powerful mechanism for enhancing specification-based test generation and highlight the adaptability of LLMs when provided with structured feedback regarding previous failures. Overall, this phase demonstrates that the prompt structure clearly impacts the outcome of the test generation phase.

5.2 Research Contributions

This dissertation contributes to the growing body of research at the intersection of software testing, formal verification, and Generative Artificial Intelligence by providing an empirical evaluation of how Large Language Models can be used to assess the reliability of feedback provided by online programming learning platforms.

The first contribution of this work is the demonstration that accepted solutions in programming learning platforms are not necessarily correct. Through the application of specification-based test generation and formal verification techniques, the proposed methodology identified multiple accepted solutions that failed to fully comply with their corresponding problem specifications. These findings provide evidence that automated evaluation systems used in educational platforms may contain blind spots and therefore benefit from additional validation mechanisms.

The second contribution is the evaluation of LLM-generated test cases as a practical mechanism for detecting inconsistencies in accepted solutions. The results show that problem specifications can be effectively leveraged as test oracles, enabling the generation of executable test suites capable of identifying defects and uncovering edge cases not covered by the platform's hidden validation process.

The third contribution is the assessment of LLM-generated formal specifications and formal proofs as a complementary validation mechanism. By automatically deriving preconditions, post-conditions, and formal proofs from natural language specifications, the proposed approach was able to corroborate most of the findings obtained through test execution, demonstrating that formal verification can serve as an additional source of evidence when evaluating software correctness.

The fourth contribution is the investigation of prompt refinement as a mechanism for improving oracle generation. Inspired by self-repair approaches found in the literature, the study demonstrates that knowledge acquired from previously identified failures can be incorporated into subsequent prompting strategies, leading to substantial improvements in test execution success rates and overall test quality.

Finally, this dissertation contributes a unified workflow that combines specification-based test generation, formal specification derivation, formal verification, and prompt refinement within a single framework. While these techniques have previously been studied independently, there is limited empirical evidence regarding their combined application. By evaluating their interaction in the context of an online programming learning platform, this work provides new insights into the benefits, limitations, and practical challenges of using LLM-based approaches to support software validation and educational platform assessment.

5.2.1 Answering Research Questions

This dissertation intends to answer the research questions that were initially posed. To directly answer each questions we have the following:

- **RQ1: To what extent can problem specifications serve as reliable test oracles?**

Answer: The results suggest that the feedback provided by programming learning platforms should not be assumed to be entirely reliable. Although the vast majority of accepted solutions analysed in this study were correct, the proposed methodology identified multiple cases in which Platform X accepted implementations that did not fully comply with their corresponding specifications. These inconsistencies were detected through both specification based test generation and LLM-generated formal proofs, which revealed overlooked edge cases and specification violations that were not captured by the platform's existing validation mechanisms.

- **RQ2: Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of test cases?**

  Answer: The findings indicate that LLM-generated test cases can effectively assist in the automated evaluation of programming learning platforms. Despite the presence of some invalid test cases and oracle-related inaccuracies, the generated test suites were capable of identifying accepted solutions that did not fully comply with their corresponding specifications. More specifically, within the *Beginner* category, i.e, where accepted solutions would generally be expected to be correct, the generated test cases revealed at least three challenges containing implementations that should not have been accepted by the platform.

- **RQ3: Can LLMs assist in automating the evaluation of programming learning platforms' feedback through the generation of formal proofs?**

  Answer: The findings indicate that LLM-generated formal proofs can assist in the automated evaluation of programming learning platforms. The generated proofs successfully identified most of the problematic implementations previously detected through specification based test generation, thereby reinforcing the consistency of the results obtained across both validation mechanisms.

- **RQ4: Could the prompt structure affect the results regarding the test cases generated by the LLM?**

  Answer: The findings clearly indicate that prompt structure has a significant impact on the quality of the generated test cases. Refining the initial prompt based on the difficulties and failure patterns identified during Phase 2 resulted in a substantial improvement in performance. More specifically, the overall Test Execution Success Rate of the challenges that previously contained failing or non-executable test cases increased by over 15 percentage points, rising from 54.86% to 70.9%.

5.3 Future Work

Due to the time constraints associated with this dissertation, the scope of the study could not be expanded further. Consequently, several opportunities remain for future researchers to build upon the knowledge and findings generated by this work.

First and foremost, a particularly promising extension would be the incorporation of mutation analysis into the proposed framework. By introducing mutated versions of the analysed algorithms, it would be possible to assess whether the generated test suites are genuinely effective at detecting defects and the output from that analysis could be used in the prompt to generate a stronger test suite. Such an evaluation would help determine whether the high number of successfully executing test cases observed throughout this study accurately reflects test quality or merely indicates that the generated tests conform to the expected input and output formats. Mutation analysis is widely recognised as one of the most effective techniques for evaluating the fault detection capability of a test suite and should therefore be considered an important addition to future iterations of this methodology whenever feasible.

Another relevant research direction concerns increasing confidence in the automatically generated preconditions, postconditions, and formal proofs. Although the formal proofs produced in this study were manually inspected to verify their final conclusions and identify obvious inconsistencies, the underlying reasoning process was not subjected to a detailed examination. Future work could therefore investigate the internal coherence of the generated proofs, analysing whether the conclusions reached are fully supported by the intermediate reasoning steps. Such an analysis would help identify potential cases of hallucination, flawed logical deductions, or incorrect interpretations of the generated conditions. Ultimately, this would provide a more comprehensive assessment of the reliability of LLM-generated formal verification artifacts.

Finally, the methodology proposed in this dissertation could be evaluated within a real world software development environment or applied to another online programming platform. Real world software systems introduce additional complexities, including larger codebases, evolving requirements, incomplete specifications, and domain specific constraints. Applying the proposed workflow in such contexts would provide valuable insights into its practical applicability, scalability, and effectiveness. Moreover, it would enable the identification of new challenges and opportunities that may not be observable within the relatively constrained environment of programming challenge repositories.

Taken together, these research directions represent natural extensions of the present work and offer promising opportunities to further advance the use of Large Language Models for software testing, oracle generation, and formal verification in the context of feedback analysis in online programming learning platforms and potentially real-world scenarios.

References

[1] E. T. Barr et al. “The Oracle Problem in Software Testing: A Survey”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525. DOI: 10.1109/TSE.2014.2372785.

[2] Concepción Batanero-Ochaíta et al. “Improving Accessibility in Online Education: Comparative Analysis of Attitudes of Blind and Deaf Students Toward an Adapted Learning Platform”. In: *IEEE Access* 9 (2021), pp. 99968–99982. DOI: 10.1109/ACCESS.2021.3095041.

[3] Arianna Blasi et al. “Translating code comments to procedure specifications”. In: New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 9781450356992. DOI: 10.1145/3213846.3213872. URL: https://doi.org/10.1145/3213846.3213872.

[4] Arda Celik and Qusay H. Mahmoud. “A Review of Large Language Models for Automated Test Case Generation”. In: *Machine Learning and Knowledge Extraction* 7.3 (2025). ISSN: 2504-4990. DOI: 10.3390/make7030097. URL: https://www.mdpi.com/2504-4990/7/3/97.

[5] Dhruva Chakravorty and Minh Tri Pham. “Evaluating the Effectiveness of an Online Learning Platform in Transitioning Users from a High Performance Computing to a Commercial Cloud Computing Environment”. In: *The Journal of Computational Science Education* 11 (1 Jan. 2020), pp. 93–99. DOI: 10.22369/issn.2153-4136/11/1/15.

[6] Yinghao Chen et al. “ChatUniTest: A Framework for LLM-Based Test Generation”. In: New York, NY, USA: Association for Computing Machinery, 2024. ISBN: 9798400706585. DOI: 10.1145/3663529.3663801. URL: https://doi.org/10.1145/3663529.3663801.

[7] Arghavan Moradi Dakhel et al. “Effective test generation using pre-trained Large Language Models and mutation testing”. In: *Information and Software Technology* 171 (2024), p. 107468. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2024.107468. URL: https://www.sciencedirect.com/science/article/pii/S0950584924000739.

[8] Elizabeth Dinella et al. “TOGA: a neural method for test oracle generation”. In: New York, NY, USA: Association for Computing Machinery, 2022. ISBN: 9781450392211. DOI: 10.1145/3510003.3510141. URL: https://doi.org/10.1145/3510003.3510141.

[9] Madeline Endres et al. “Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?” In: 1.FSE (2024). DOI: 10.1145/3660791. URL: https://doi.org/10.1145/3660791.

[10] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. "ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance". In: *IEEE Transactions on Software Engineering* 51.1 (2025), pp. 305–319. DOI: 10.1109/TSE.2024.3519159.

[11] Soneya Binta Hossain and Matthew B. Dwyer. "TOGLL: Correct and Strong Test Oracle Generation with LLMs". In: IEEE Press, 2025. ISBN: 9798331505691. DOI: 10.1109/ICSE55347.2025.00098. URL: https://doi.org/10.1109/ICSE55347.2025.00098.

[12] Soneya Binta Hossain, Raygan Taylor, and Matthew Dwyer. "Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation". In: 2.FSE (2025). DOI: 10.1145/3729354. URL: https://doi.org/10.1145/3729354.

[13] Abderrahman Jaize et al. "Evaluating the Effectiveness of an Online Learning Platform - A Study Of A Google Cloud Learning System". In: *2020 6th IEEE Congress on Information Science and Technology (CiSt)*. 2020, pp. 236–241. DOI: 10.1109/CiSt49399.2021.9357311.

[14] Nitesh Kumar Jha, Plaban Kumar Bhowmik, and Kaushal Kumar Bhagat. "Usability Evaluation of an Online Inquiry-Based Learning Platform for Computational Thinking (CT-ONLINQ)". In: *2023 IEEE International Conference on Advanced Learning Technologies (ICALT)*. 2023, pp. 182–186. DOI: 10.1109/ICALT58122.2023.00059.

[15] Jack Johnson et al. "Generating Failure-Based Oracles to Support Testing of Reported Bugs in Android Apps". In: *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2025, pp. 2196–2208. DOI: 10.1109/ASE63991.2025.00182.

[16] Shaker Mahmud Khandaker et al. "AugmenTest: Enhancing Tests with LLM-Driven Oracles". In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2025, pp. 279–289. DOI: 10.1109/ICST62969.2025.10988926.

[17] Brahma Reddy Korraprolu, Pavitra Pinninti, and Y. Raghu Reddy. "Test Case Generation for Requirements in Natural Language - An LLM Comparison Study". In: New York, NY, USA: Association for Computing Machinery, 2025. ISBN: 9798400714245. DOI: 10.1145/3717383.3717389. URL: https://doi.org/10.1145/3717383.3717389.

[18] Na Li et al. "AI-Driven Test Case Generation Based on DeepSeek-Chat Large-Language Model". In: *2025 5th International Symposium on Computer Technology and Information Science (ISCTIS)*. 2025, pp. 1–4. DOI: 10.1109/ISCTIS65944.2025.11065082.

[19] Richard Wing Cheung Lui, Michel Nga Yin Dik, and Philip Tin Yun Lee. "Evaluation of Online Learning Platforms for Interactive Self-paced Learning of Container-based Software Development". In: *2022 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*. 2022, pp. 54–58. DOI: 10.1109/TALE54877.2022.00017.

[20] Noble Saji Mathews and Meiyappan Nagappan. "When AI-Generated Unit Tests Validate Bugs: The Risk of Faulty Assertions". In: *IEEE Software* 43.1 (2026), pp. 98–104. DOI: 10.1109/MS.2025.3597574.

[21] Facundo Molina, Nazareno Aguirre, and Alessandra Gorla. "State Field Coverage: A Metric for Oracle Quality". In: *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2025, pp. 2707–2719. DOI: 10.1109/ASE63991.2025.00222.

[22] OpenAI. *Introducing GPT-OSS*. 2025. URL: https://openai.com/index/introducing-gpt-oss/.

[23] Harijanto Pangestu and Marisa Karsen. "Evaluation of usability in online learning". In: *2016 International Conference on Information Management and Technology (ICIMTech)*. 2016, pp. 267–271. DOI: 10.1109/ICIMTech.2016.7930342.

[24] Warren Panizales et al. "Predicting Student Dropout Rates in Online Education Platforms utilizing Naive Bayes Algorithm". In: *2025 16th International Conference on E-Education, E-Business, E-Management and E-Learning (IC4e)*. 2025, pp. 425–429. DOI: 10.1109/IC4e65071.2025.11075311.

[25] Severin Primbs, Benedikt Fein, and Gordon Fraser. "AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model". In: *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. 2025, pp. 12–23. DOI: 10.1109/AST66626.2025.00008.

[26] Ramneet, Deepali Gupta, and Mani Madhukar. "Operational Challenges in Online Self-Learning Education Adoption". In: *2021 6th International Conference on Signal Processing, Computing and Control (ISPCC)*. 2021, pp. 51–55. DOI: 10.1109/ISPCC53510.2021.9609447.

[27] Ekta Sinha and Kenneth Bagarukayo. "Online Education in Emerging Knowledge Economies:Exploring factors of motivation, de-motivation and potential facilitators; and studying the effects ofdemographic variables". In: *The International Journal of Education and Development using Information and Communication Technology* 15 (June 2019), pp. 5–30.

[28] Dimitris Spiliotopoulos et al. "MOOC Accessibility from the Educator Perspective". In: *HCI International 2020 – Late Breaking Papers: Universal Access and Inclusive Design*. Ed. by Constantine Stephanidis et al. Cham: Springer International Publishing, 2020, pp. 114–125.

[29] Ruofan Yang, Xianghua Xu, and Ran Wang. "TestLoter: A logic-driven framework for automated unit test generation and error repair using large language models". In: *Journal of Computer Languages* 84 (2025), p. 101348. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2025.101348. URL: https://www.sciencedirect.com/science/article/pii/S2590118425000346.

[30] Zhuoying Yang and Lei Wang. "IntelliUnitGen: A Unit Test Case Generation Framework Based on the Integration of Static Analysis and Prompt Learning". In: *IEEE Access* 13 (2025), pp. 177760–177784. DOI: 10.1109/ACCESS.2025.3615990.

[31] Quanjun Zhang et al. "Exploring Automated Assertion Generation via Large Language Models". In: 34.3 (2025). ISSN: 1049-331X. DOI: 10.1145/3699598. URL: https://doi.org/10.1145/3699598.

Appendix A

Formal proof

An example of a generated Formal Proof can be found in the page below.

Formal Proof: Hoare-Logic Correctness of the Counting Program

1. What Has to Be Shown

We are asked to prove that the following Python program

```
n = int(input())
in_ = 0
out = 0
for i in range(n):
    x = int(input())
    if 10 <= x <= 20:
        in_ += 1
    else:
        out += 1
print(f'{in_}')
print(f'{out}')
```

(In the source text the counter that Python reserves the keyword "in" for is written simply as "in"; here it is rendered as in_ to keep it syntactically valid, but it denotes the same variable referred to as 'in' throughout the proof.)

satisfies the Hoare-logic specification

{ Pre } P { Post }

Pre-condition

Pre $\equiv$ $N \in \mathbb{N} \land N < 10\,000$

$\land \forall i \, (1 \le i \le N \Rightarrow X_i \in \mathbb{Z} \land -10^7 < X_i < 10^7)$

Post-condition

Post $\equiv$ $I = [10, 20]$

$\land$ $in = \#\{ i \mid 1 \le i \le N \land X_i \in I \}$

$\land$ $out = N - in$

$\land$ the program prints two lines: the first is the value of in, the second is the value of out.

(The interval in the source code is [10, 20]; we therefore replace the generic I = [0, 100] that appears in the general statement of the problem by the concrete interval used by the program. The proof proceeds exactly the same way for any closed interval.)

Thus we must show that, starting from a state that satisfies Pre, after the execution of the whole program the state satisfies Post.

2. Hoare-Logic Rules We Shall Use

Rule	Meaning
Assignment	$\{ P[E/x] \}\ x := E\ \{ P \}$
Sequence	$\{ P \}\ S_1\ \{ Q \},\ \{ Q \}\ S_2\ \{ R \} \Rightarrow \{ P \}\ S_1; S_2\ \{ R \}$
Conditional	$\{ P \land B \}\ S_1\ \{ Q \},\ \{ P \land \neg B \}\ S_2\ \{ Q \} \Rightarrow \{ P \}$ if B then S_1 else S_2 $\{ Q \}$

Rule	Meaning
While (partial correctness)	If { Inv ∧ B } S { Inv } then { Inv } while B do S { Inv ∧ ¬B }, with Inv an invariant.
Consequence	From P ⇒ P′ and { P′ } S { Q′ } and Q′ ⇒ Q, we obtain { P } S { Q }.

We will apply these rules to each statement of the program in turn.

3. Decomposing the Program

Let us rename the program variables so that they match the symbols of the specification:

Python variable	Specification symbol
n	N
x	X_i (the value read in the i-th iteration)
in	in
out	out

The program consists of the following blocks:

1. **Read N**

```
n = int(input())
```

2. **Initialise the counters**

```
in = 0
out = 0
```

3. **Counting loop**

```
for i in range(n):      # i runs from 0 to N-1
    x = int(input())
    if 10 <= x <= 20:
        in += 1
    else:
        out += 1
```

4. **Printing**

```
print(f'{in}')
print(f'{out}')
```

4. Proof of the First Three Blocks

4.1 Reading N

The input operation does not change any program variable; it merely binds n to the first integer on the input stream. By the definition of the pre-condition we already know that this first integer belongs to ℕ and is less than 10 000, therefore after the assignment we have

$\{ \text{Pre} \}\ \ n := N\ \ \{ N \in \mathbb{N} \wedge N < 10\,000 \wedge \Phi \},$

where Φ abbreviates the universal condition on the subsequent inputs. Thus the post-condition of this statement is exactly the pre-condition of the remaining program.

4.2 Initialising the Counters

Using the assignment rule twice:

$\{ N \in \mathbb{N} \wedge N < 10\,000 \wedge \Phi \}\ \ in := 0\ \ \{ in = 0 \wedge \dots \}$

$\{ in = 0 \wedge \dots \}\ \ out := 0\ \ \{ in = 0 \wedge out = 0 \wedge \dots \}$

Consequently, after the two initialisations we have the invariant

$Inv_0 \equiv (in = 0) \wedge (out = 0) \wedge (N \in \mathbb{N} \wedge N < 10\,000 \wedge \Phi).$

5. Loop Invariant for the Counting Loop

We use a while-style invariant because a for loop can be rewritten as a while loop with an explicit counter. Let

```
i := 0                      # loop counter
while i < N:
    x := read()
    if 10 <= x <= 20:
        in := in + 1
    else:
        out := out + 1
    i := i + 1
```

5.1 Definition of the Invariant

$Inv(k) \equiv 0 \le k \le N$

$\wedge\ in_k = \#\{ j \mid 1 \le j \le k \wedge X_j \in [10, 20] \}$

$\wedge\ out_k = k - in_k,$

where in_k and out_k denote the values of the counters after the first k iterations — i.e., when the loop variable i equals k. Informally: after having processed the first k numbers, we have already counted exactly those that belong to the interval, and the rest fall in the complementary count.

5.2 Initialisation (base case k = 0)

Before the loop starts we have i = 0, in = 0, out = 0. Substituting k = 0 in Inv(k) we obtain

$0 \le 0 \le N \wedge in = 0 \wedge out = 0,$

which is precisely the state established by the initialisation block. Hence

$\{ Inv_0 \}\ \ i := 0\ \ \{ Inv(0) \}.$

5.3 Preservation

Assume Inv(k) holds at the beginning of an iteration and that the loop guard i < N (i.e. k < N) is true. We must show that after executing the body, the state satisfies Inv(k+1).

Reading x does not affect the counters, therefore the invariant still holds with the same in_k and out_k.

Conditional:

```
if 10 <= x <= 20:
    in := in + 1
else:
    out := out + 1
```

Let

$$c = [\ 10 \le x \le 20\ ] \in \{0, 1\}$$

(i.e. the indicator of the condition). Then the two branches can be written compactly as

$$in := in + c, \quad out := out + (1 - c).$$

Using the assignment rule we obtain

$$\{ Inv(k) \wedge c = 1 \}\ \ in := in + 1\ \ \{ in_{k+1} = in_k + 1 \wedge out_{k+1} = out_k \}$$

$$\{ Inv(k) \wedge c = 0 \}\ \ out := out + 1\ \ \{ in_{k+1} = in_k \wedge out_{k+1} = out_k + 1 \}.$$

In both cases we have

$$in_{k+1} = in_k + c, \quad out_{k+1} = out_k + (1 - c),$$

and by definition of c,

$$c = [\ X_{k+1} \in [10, 20]\ ].$$

Therefore

$$in_{k+1} = \#\{ j \mid 1 \le j \le k \wedge X_j \in [10, 20] \} + [\ X_{k+1} \in [10, 20]\ ]$$

$$= \#\{ j \mid 1 \le j \le k+1 \wedge X_j \in [10, 20] \},$$

and similarly

$$out_{k+1} = (k + 1) - in_{k+1}.$$

Thus, after the conditional, the invariant Inv(k+1) holds.

Increment of the loop counter:

```
i := i + 1
```

This adds 1 to i, turning the value k into k+1. The invariant Inv(k+1) already mentions k+1, therefore the assignment preserves it.

Consequently

$$\{ Inv(k) \wedge i < N \}\ \ body\ \ \{ Inv(k + 1) \}.$$

5.4 Termination

When the loop guard becomes false we have i = N. Applying the while rule we obtain

$$\{ Inv(0) \}\ \ \text{while } i < N \text{ do body}\ \ \{ Inv(N) \wedge i \ge N \}.$$

Since i never exceeds N, the condition i ≥ N simplifies to i = N. Thus, after the loop, we know

$$Inv(N) \equiv in = \#\{ j \mid 1 \le j \le N \wedge X_j \in [10, 20] \} \wedge out = N - in.$$

6. The Printing Statements

We have proved that after the loop the program state satisfies exactly the numerical part of the post-condition. The two print statements are simple output commands; they do not modify any variable. Using the consequence rule we can strengthen the post-condition of the loop with the additional requirement that the two lines are printed:

```
print(f'{in}')
print(f'{out}')
```

Both statements are instances of the rule

$$\{ P \}\ \text{print}\ e\ \{ P \},$$

because printing is observationally pure with respect to the logical state. Consequently the final Hoare triple becomes

$\{ \text{Pre} \}\ P\ \{ \text{Post} \}$.

7. Summary of the Proof

Program fragment	**Hoare triple derived**
n = int(input())	$\{ \text{Pre} \}\ n := N\ \{ N \in \mathbb{N} \land N < 10\,000 \land \Phi \}$
in = 0; out = 0	$\{ N \in \mathbb{N} \land N < 10\,000 \land \Phi \}$ in := 0; out := 0 $\{ \text{Inv}(0) \}$
counting loop	$\{ \text{Inv}(0) \}$ while i < N do body $\{ \text{Inv}(N) \}$ (proved via loop invariant Inv(k))
printing	$\{ \text{Inv}(N) \}$ print …; print … $\{ \text{Post} \}$
Composition	By the sequence rule, the four rows above combine to give the overall triple $\{ \text{Pre} \}\ P\ \{ \text{Post} \}$.

Thus, assuming the pre-condition holds — a non-negative integer N smaller than 10 000, followed by exactly N integers each in the range $(-10^7, 10^7)$ — the program terminates and, on termination:

- in equals the number of input values that lie in the interval [10, 20];
- out equals N − in;
- the two required lines are printed.

Hence the program is **correct** with respect to the Hoare-logic specification given in the problem statement.