Trustworthy AI Feedback

A Hybrid Approach to Automated Feedback: Combining Rule-Based Reasoning with Generative AI in Ask-Elle

Master's thesis in Computer science – Algorithms, Languages and Logic

ARASH AMIRY & NILS BENGTSSON SVANSTEDT

DEPARTMENT OF COMPUTER SCIENCE

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Trustworthy AI Feedback

A Hybrid Approach to Automated Feedback: Combining Rule-Based Reasoning with Generative AI in Ask-Elle

ARASH AMIRY & NILS BENGTSSON SVANSTEDT

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Trustworthy AI Feedback
A Hybrid Approach to Automated Feedback:
Combining Rule-Based Reasoning with Generative AI in Ask-Elle
Arash Amiry & Nils Bengtsson Svanstedt

Supervisor: Alex Gerdes, Department of Computer Science and Engineering
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Division of Computer Science
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Trustworthy AI Feedback
A Hybrid Approach to Automated Feedback:
Combining Rule-Based Reasoning with Generative AI in Ask-Elle
Arash Amiry & Nils Bengtsson Svanstedt
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Ask-Elle is an intelligent tutoring system for learning functional programming in Haskell, providing feedback through strategy-based model tracing and property-based testing. However, the system fails to offer meaningful guidance in cases where a student's program deviates from model solutions. This thesis investigates integrating generative AI into Ask-Elle to address this limitation. The proposed hybrid architecture uses a large language model to complete the student's partial code, which is then validated by Ask-Elle's existing compiler and property-based tests before any hint is shown to the student. Three models were evaluated, Claude Sonnet 4.6, GPT-5.3-Codex, and GPT-4.1-nano, with the two larger models achieving success rates of 88–97%. GPT-5.3-Codex was selected as the final model based on both technical performance and a blind pedagogical evaluation of hint quality. A user study indicated that AI-generated hints provided higher educational value than the system's existing feedback. While code correctness is reliably enforced by the chosen architecture, occasional hallucinations in natural language hints remain an open challenge.

Keywords: intelligent tutoring systems, generative AI, automated feedback, functional programming, Haskell, property-based testing, model tracing.

Acknowledgements

A special thank you goes to Alex Gerdes for being an exceptionally supportive supervisor. We truly appreciate his patience and the inspiring brainstorming sessions that helped us navigate the more complex parts of this project.

Arash Amiry, Nils Bengtsson Svanstedt, Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AAII	Artificial Analysis Intelligence Index
AI	Artificial Intelligence
API	Application Programming Interface
DL	Deep Learning
ED	Edit Distance
GenAI	Generative Artificial Intelligence
GHC	Glasgow Haskell Compiler
GPQA	Graduate-Level Google-Proof Q&A
ITS	Intelligent Tutoring System
LLM	Large Language Model
LVM	Lazy Virtual Machine
ML	Machine Learning
MMLU	Massive Multitask Language Understanding
NED	Normalized Edit Distance
RQ	Research Question
UHA	Unified Haskell Architecture

Contents

1

Introduction

Over the past decades, feedback mechanisms have become increasingly diverse, with new techniques emerging to deliver more effective and pedagogically useful guidance to students [1]. Intelligent tutoring systems (ITS) is a class of educational software that incorporate computational techniques to provide adaptive and personalized learning experiences. Unlike earlier forms of computer-assisted instruction, ITS dynamically track students' progress, and provide tailored feedback to individual need [2].

Digital tools for education, such as artificial intelligence (AI), are increasingly used to enhance and personalize teaching and learning processes [3]. Advances in generative artificial intelligence (GenAI), especially large language models (LLMs), have opened new opportunities for creating more personalized feedback for students. However, integrating such LLMs into educational tools introduces challenges related to the reliability and accuracy of their explanations.

Ask-Elle is an ITS designed to assist students with learning functional programming in Haskell [4]. An ITS imitates human tutors to provide automated feedback, which Ask-Elle does through analyzing student solutions using strategy-based model tracing and property-based testing. A core feature of Ask-Elle is its support for stepwise development, which allows students to build programs incrementally rather than only submitting a final solution. This process utilizes holes (placeholders denoted by ? in the code) to represent missing expressions. By refining these holes one step at a time, students can receive guidance on intermediate and incomplete code. Although the feedback mechanisms are pedagogical and accurate, they also have limitations. When a student's solution deviates from predefined model solutions, the system can still report compiler errors or find counterexamples via property-based testing. However there are specifically two cases (the "Undecided" states) where Ask-Elle fails to provide meaningful feedback. The first case involves students who follows a valid but unrecognized strategy that passes all tests. The second case occurs when the student's solution does not match with any model solution and the majority of test cases are discarded, in almost all cases due to the program being undefined at too many places.

Ask-Elle's inability to provide guidance in the first case stems from two primary hurdles: incomplete strategy coverage and the limits of program transformation. Since the set of model solutions in Ask-Elle is finite, the strategies derived from them are finite as well, meaning they can never exhaustively cover every valid implementation

of a given programming task. While a program may pass all property-based tests, such testing cannot guarantee absolute correctness. It is limited by input coverage and is primarily designed to prove that a program is incorrect by identifying a counterexample. If a student's solution passes all tests but cannot be matched to a model solution, Ask-Elle cannot definitively confirm its correctness. This is because program transformation is bound by a fundamental limitation: we can never mathematically prove two arbitrary programs to be functionally equivalent. Consequently, such submissions remain in a state where total correctness cannot be ensured.

The underlying problem for Ask-Elle's inability to provide guidance is the interaction between property-based testing and the presence of holes within incomplete programs. When Ask-Elle utilizes QuickCheck to evaluate a student's program (which is unrecognized by Ask-Elle) against defined properties, any test case that encounters an undefined hole during execution is discarded. When the system discards too many test cases, it lacks the necessary data to either find a counterexample or confirm that the properties are satisfied. However, even if a student refines their program to the point that it successfully passes all properties while remaining unrecognized by Ask-Elle, the absence of a matching strategy or the failure of program transformation still leaves total correctness unconfirmed.

To illustrate how Ask-Elle's components work in practice, consider an exercise where a student must implement a function `myreverse` to reverse the order of a list. This example will also highlight where Ask-Elle fails to provide meaningful feedback when a student's solution drifts too far from any known strategy. A student might begin with the following code:

```
myreverse [] = []
myreverse (x:xs) = ?
```

After the student submits the code, Ask-Elle compiles the code, generates an abstract syntax tree, normalizes it and then uses strategy-based model tracing to verify that this intermediate step aligns with one of the model solutions. The model solution used for this comparison might include pedagogical annotations, such as DESC and FC to guide the student with different level of detail:

```
{-# DESC Use explicit @recursion. #-}
myreverse [] = []
{-# FC @append can be used to put an item at the end of a list. #-}
myreverse (x:xs) = myreverse xs ++ [x]
```

If the student proceeds by filling the remaining hole with an incorrect expression, such as `myReverse (x:xs) = myReverse xs`, the system utilizes property-based testing to detect the error:

```
propModel f = counterexample "You have not reversed correctly" $
    \xs -> f xs == reverse xs
```

When this property is falsified by a generated test case, Ask-Elle reports a concrete counterexample to the student. For this specific error, the system would find that for an input like [1], the student's code returns [ ] while the expected result was [1].

After receiving the counterexample, the student tries a different approach, rewriting the function using an accumulator pattern:

```
myreverse = reverse' []
    where
        reverse' acc [] = ?
        reverse' acc ? = ?
        reverse' ? ?
```

Assuming the only existing model solution for this example is the one shown earlier, submitting this code leaves the student with no meaningful feedback, only the message: *"You have drifted from the strategy in such a way that we can not help you any more"*.

It is worth noting that model solutions already exist for exercises in Ask-Elle, however, in practice it is difficult to ensure that it has access to all high-quality solutions. This thesis proposes a practical fix: if Ask-Elle gets stuck and cannot provide meaningful feedback, particularly when a submission contains too many holes, the system hands the problem over to an LLM. The LLM looks at the student's incomplete code and figures out a way to fill in the gaps, producing a complete, working solution. However, that AI-completed code is never shown to the student. Instead, Ask-Elle runs its own tests on it first. Only if the completed code actually passes the compilation and testing stage does the system then ask the LLM to write a short, helpful hint for the student, pointing them toward their next step without giving the answer away. This kind of approach, to generate, verify, then advise, means the student only receives feedback that is grounded in code that has been tested to work.

1.1 Aim

The aim of this thesis is to investigate whether integrating GenAI into Ask-Elle enhances the quality and coverage of automated feedback for students learning functional programming. The work seeks to combine the flexibility of AI generated feedback with the reliability of Ask-Elle's existing strategy-based model tracing and property-based testing.

A central objective was the design and evaluation of a hybrid workflow. This work resulted in the development of a prototype and an evaluation demonstrating that the combined system provides more useful feedback than the standalone version of Ask-Elle, while maintaining high reliability.

1.2 Scope

This thesis focuses on the integration of GenAI into the existing framework of Ask-Elle, limited to feedback generation for functional programming exercises written in Haskell. Specifically, it targets student submissions where the presence of holes causes too many test cases to be discarded, leaving Ask-Elle unable to provide meaningful feedback. Submissions where all property-based tests pass are excluded, as no concrete behavioral violations can be identified. Submissions that already trigger effective feedback mechanisms, such as those that fail to compile, match a model solution, or produce a counterexample, are excluded from further consideration in this thesis.

The project addresses the design and prototyping of workflows in which AI generated feedback is combined with Ask-Elle's strategy-based model tracing and property-based testing. The evaluation is limited to assessing the correctness, reliability, and educational value of the combined system compared to Ask-Elle's current system. Ask-Elle's underlying architecture, compiler infrastructure, or core analysis algorithms will neither be modified nor redesigned during the project.

1.3 Research Questions and Challenges

The main goal of this thesis was to improve Ask-Elle's feedback capabilities by integrating GenAI, while ensuring the reliability of its output. To structure this goal into a research direction, the work was guided by the following research questions:

- **RQ1:** How was GenAI integrated into Ask-Elle's existing strategy-based and property-based feedback mechanisms in a way that complements its rule-based reasoning? This question is addressed in Section 4.2.

- **RQ2:** How was AI generated feedback validated by Ask-Elle's existing tools to ensure correctness and reliability before it is shown to students? The question is addressed in Section 3.1.1

- **RQ3:** To what extent did the combined system of Ask-Elle and GenAI improve the quality and educational value of feedback compared to Ask-Elle alone? This question is addressed in Section 4.7.

- **RQ4:** How reliable was the AI-enhanced Ask-Elle prototype in generating code that satisfies the defined acceptance criteria (mentioned in Section 3.1.1) and producing non-hallucinated feedback? This question is addressed in Section 4.2 and 4.6.1.

These research questions frame the goals of the project. The work involved designing an integration of GenAI into Ask-Elle's analysis pipeline, prototyping mechanisms that filter, validate or adapt generated feedback using Ask-Elle's model tracing and

property-based testing, and evaluating the resulting system using a combination of experiments and user studies.

Our approach is to design a prompt engineering system that instructs a GenAI to generate two distinct outputs for every student submission:

1. A **complete, testable code solution** intended for machine validation.
2. A **pedagogical hint** in natural language intended for the student.

The central principle of our architecture is that the pedagogical hint (2) is only shown to the student if and only if the complete code solution (1) is first verified as correct by our property-based tests. This ensures the advice given by the GenAI is grounded in demonstrably correct code, reducing risk of hallucination.

The primary challenge is tackling the unreliability of GenAI, as it is prone to "hallucination". Hallucination is a term that describes when AI generates plausible sounding incorrect information [5]. This challenge is especially important to tackle as the Ask-Elle tool is to be primarily used by people with no prior Haskell experience, who are less likely to spot incorrect code. Even small inaccuracies can cause substantial confusion or reinforce misconceptions.

Another important challenge is to balance the pedagogical usefulness with computational performance. Generating and validating useful feedback comes at the expense of additional computational overhead. Unlike Ask-Elle's existing feedback mechanisms, the hybrid pipeline introduces several additional processing steps. Each time a student submission is classified as Discarded, the system must construct a prompt, send it to an LLM, wait for a response, extract the generated code, compile it, and run the full property-based test suite against it.

2
Background

Ask-Elle analyzes student solutions written in Haskell, a purely functional programming language designed to emphazise declarative programming, static typing, and immutability. In Haskell, programs are often constructed by composing functions, and the computation is performed through the evaluation of expressions rather than sequences of commands. The language features strong static typing, allowing compilers to detect type errors that can be used to provide feedback to students. These characteristics influence how Ask-Elle analyzes student submissions and generates feedback.

The following sections outline the technical foundation of this work, moving from Haskell compilers, model tracing and QuickCheck, to the generative AI technologies and the Ask-Elle system architecture.

2.1 Helium

Helium is a compiler specifically designed to deliver high quality type error messages particularly for beginner programmers of Haskell [6]. Internally, the Helium compiler is composed of a modular pipeline that translates Haskell source code through numerous stages [7]. The source code is first parsed into the Unified Haskell Architecture (UHA), a representation that closely mirrors Haskell's concrete syntax. This ensures that error messages refer to the code exactly as written by the user. Static checks are performed on the UHA to identify structural mistakes and undefined identifiers. Following analysis, the program is translated into Core, an untyped intermediate lambda calculus. Finally, the Core code is converted into instruction files for execution on the Lazy Virtual Machine (LVM). The pipeline is shown in Figure 2.1.

The compiler identifies mistakes in the structure or logic of the code, and provides precise syntax and type error diagnosis stages [7]. A key feature of its diagnostic capability is the provision of exact positional data, including both line and column numbers, to assist students in pinpointing the origin of an error. For example, Helium effectively identifies the illegal nesting or omission of brackets. Consider the following code:

```
test :: [(Int, String)]
test = [(1, "one"), (2, "two"), (3, "three")
```

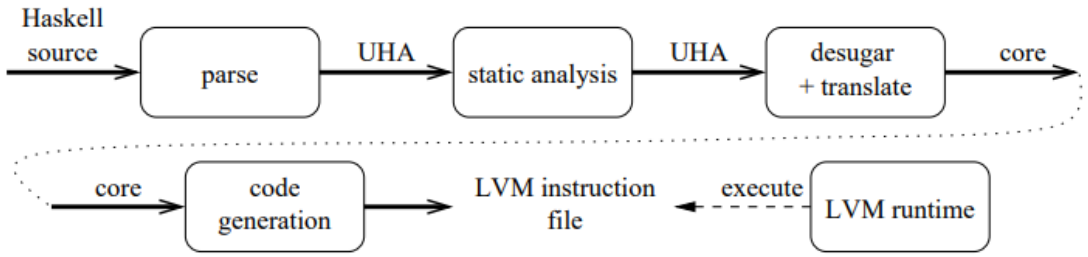

Figure 2.1: Architectural pipeline of the Helium compiler.

In this instance, the programmer has omitted the closing bracket for the list, a mistake that Helium's research identifies as a very common lexical error [7]. Helium's lexical analyzer maintains an internal stack to track the nesting of all parentheses, braces, and brackets, which allows the compiler to produce a clear message, such as:

```
(2,8): Bracket '[' is never closed
```

To provide clearer type error diagnostics than traditional Haskell compilers, Helium utilizes a constraint-based type inferencer [7]. A primary strength of the Helium type checker is that it preserves the entire type inference graph of a program. Helium can produce a clear and precise error message by reviewing the type inference graph. For instance, if a student provides the arguments of a function in the incorrect order.

```
test = map [1..10] even
```

In this case, the student has accidentally swapped the list and the predicate function for map. Because Helium keeps track of the entire inference context, it can identify that the arguments have been reordered and suggest a "probable fix" such as "re-order arguments" [7].

```
(1,8): Type error in application
 expression     : map [1..10] even
 term           : map
 type           : (a -> b) -> [a] -> [b]
 does not match: [Int] -> (Int -> Bool) -> c
 probable fix   : re-order arguments
```

Within Ask-Elle, Helium serves as the first stage of the submission analysis pipeline. When a student submits their code, it is passed through Helium before any further analysis takes place. Helium's primary role at this stage is to verify that the submission is syntactically and type-correct, and to produce clear error messages if it is not. Additionally, Ask-Elle has extended the compiler to manage incomplete programs [4].

2.2 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is a state-of-the-art open source compiler and interactive environment for the functional language Haskell [8]. While traditional compilers are primarily used to transform source code into executable binaries, GHC's architecture also allows it to be used for the evaluation of Haskell expressions with an interactive interpreter called GHCi [9].

In the context of Ask-Elle, GHC provides the necessary infrastructure to test properties and to evaluate student solutions [4]. This infrastructure can be accessed through the GHC API, which exposes the GHC's functionality to external applications [9]. The architecture of GHC allows it to be integrated as a library within software systems. This has facilitated the development of diverse tools, including `hint`, which provides an API for the evaluation of Haskell source code. Within Ask-Elle, the `hint` library manages interpreter sessions by loading student solutions and exercise data while restricting the environment to core libraries like the `Prelude` and `QuickCheck`. The student's code is then passed to the GHC interpreter via the `hint` library, which first compiles the submission to verify that it is valid. Once compilation is successful, it then executes property-based tests to either uncover counterexamples in the student's logic or verify the functional correctness of the solution.

2.3 Model Tracing

Model tracing tutors represent an effective category of intelligent learning environments [10]. The defining characteristic of these systems is the expert model, which is composed of rules that encapsulate the procedural knowledge required to solve problems. The mechanism of verifying student actions against the expert model is known as model tracing. Depending on the pedagogical strategy, the results of this check can be delivered as immediate feedback to provide real-time guidance or held in reserve until the student explicitly requests feedback. Model tracing aims to decode the student's internal problem-solving logic, using this comprehensive cognitive insight to manage the tutoring process [11]. This allows the system to provide personalized instruction based on a deep understanding of the student's current mental model.

Ask-Elle's understanding of a student's intent is achieved through a normalization and a matching process. When a student submits code, the compiler produces an abstract syntax tree which is then normalized to remove irrelevant syntactic variations. The result is then compared against the intermediate states described by the strategies derived from model solutions. A match indicates that the student is following a known valid solution path, after which feedback can be provided. Ask-Elle's understanding of the student's solution is therefore bounded by the set of model solutions it has been explicitly provided.

2.4 QuickCheck

QuickCheck is a Haskell library for testing code using property-based testing, a tool introduced by Claessen and Hughes [12]. Property-based testing is the process of defining properties that some software component should satisfy. These properties are validated through the automatic generation of test cases based on generators for the data types. Generators are functions for how to generate a random value for some data type. The strength of property-based testing is that it facilitates large scale testing without the burden of writing tests by hand. Furthermore, removing the human element also removes the human bias, potentially creating more diverse test cases.

The property below illustrates a common verification rule: the length of two joined lists must always equal the sum of their individual lengths.

```
prop_lengthAppend :: [Int] -> [Int] -> Bool
prop_lengthAppend xs ys = length (xs ++ ys) == length xs + length ys
```

A feature of QuickCheck is its shrinking mechanism [13]. When a failing test case is found, QuickCheck does not report the original randomly generated input. Instead, it attempts to reduce the counterexample to its simplest possible form by systematically shrinking the input while still preserving the failure. For example, if a property fails on a randomly generated list of 47 elements, QuickCheck will attempt to find the smallest list that still triggers the same failure.

This has direct pedagogical implications in the context of Ask-Elle. Rather than providing a student with a large failing input that may obscure the underlying mistake, the shrunk counterexample points as precisely as possible to the error in their code.

2.5 GenAI and LLMs

GenAI is situated as a specialized and advanced subset within a hierarchy of computational disciplines [14]. This relationship is structured as a series of nested subfields, where each layer serves as the technological foundation for the next.

AI
Serving as the overarching domain, AI encompasses computational algorithms capable of performing tasks that normally require human intelligence, such as natural language understanding, pattern recognition, and decision-making.

Machine Learning
Machine Learning (ML) serves as a specialized branch of AI dedicated to creating systems that solve problems independently by learning from data. This approach eliminates the requirement for explicit programming for every possible situation, allowing the algorithm to derive its own logic from the information it processes.

Deep Learning
Deep Learning (DL) represents an advanced subset of ML that employs multi-layered neural networks to process high-dimensional data. These computational models, inspired by the structure of the human brain, are capable of autonomously detecting complex correlations and patterns in large datasets.

GenAI
Functioning as a niche subset of DL, GenAI utilizes DL techniques to create new content instead of merely discriminating between existing data points. By learning inherent data structures, the system produces probabilistic outputs, including source code and natural language, that satisfy specific user instructions.

LLMs represent a specialized application of GenAI, specifically engineered to process and generate human-like language with high contextual relevance [14]. LLMs are built on the Transformer architecture, a specific type of neural network that utilizes self-attention mechanisms to understand long-range dependencies in data. LLMs do not produce deterministic or replicable results, which means outputs are inherently probabilistic. Because of this probabilistic nature, an LLM may generate different, though still valid, responses to the exact same prompt.

The LLMs evaluated in this study have all been benchmarked using the Artificial Analysis Intelligence Index (AAII). AAII is an independent metric that aggregates LLM performance across ten testing suites spanning four equally weighted categories: agentic workflows, code generation, general reasoning, and scientific reasoning [15].

GPT-5.3-Codex
GPT-5.3-Codex is a code-specialized LLM developed by OpenAI, positioned within the GPT-5 family [16]. The model is priced at $1.75 per million input tokens and $14.00 per million output tokens [17]. Using the highest reasoning effort (xhigh), GPT-5.3-Codex achieved an AAII of 54, currently placing it at 9th place out of 146 models in the same category (proprietary reasoning models priced above $1.00 per million tokens) [18].

GPT-4.1-nano
GPT-4.1-nano is another LLM developed by OpenAI, which represents the most resource-efficient tier within the GPT-4.1 model family, specifically optimized for high-throughput tasks and minimal latency [19]. The model is priced at $0.10 per million input tokens and $0.40 per million output tokens [17]. GPT-4.1-nano achieves an AAII score of 11 [20]. This score places the model at 58th place out of 84 models in the same category (non-reasoning proprietary models priced between $0.15 to $1.00 per million tokens).

Claude Sonnet 4.6
Claude Sonnet 4.6 is an LLM developed by Anthropic [21]. The price per million

tokens is $3.0 for input and $15 for output [22]. Claude Sonnet 4.6 (Adaptive Reasoning, Max Effort) achieved an AAII of 52, currently placing it at 14th place out of 146 models in the same category (proprietary reasoning models priced above $1.00 per million tokens) [23].

2.6 Ask-Elle

The ITS Ask-Elle, supports students through automated, strategy-based feedback [4]. The system combines strategy-based model tracing, which tracks how students approach programming problems, with property-based testing, which evaluates program behavior against formally defined properties. The latter technique builds upon QuickCheck. These mechanisms allow Ask-Elle to verify the correctness of student solutions and provide structured feedback aligned with pedagogical goals.

Ask-Elle analyzes stepwise program development, where students gradually refine incomplete programs that initially contain placeholders representing missing expressions, so-called "holes" [4]. A hole functions as a syntactic stand-in for an expression that has not yet been written, allowing the program to remain structurally valid while still unfinished. Ask-Elle does not require full solutions. Instead it evaluates each incremental refinement until it is a full solution, enabling feedback even on programs that are incomplete. Figure 2.2 illustrates how Ask-Elle reveals possible strategies the student can follow and the problem description for the exercise. The editor in which the student writes the solution to the exercise is shown in Figure 2.3.

The solution that the student submits is compiled by both Helium and GHC, to report syntax errors and type errors, even in the presence of holes [4]. Helium is a compiler that is intended to provide clearer and more pedagogically useful error messages than the popular GHC compiler [7]. After parsing and type checking, the student solution is then transformed through a series of normalization steps, to remove irrelevant syntactic variations [4]. The normalization process ensures that programs that contain some different but semantically equivalent constructions, can be compared against model solution strategies. For example, a variable named `acc` and one named `accumulator` are semantically identical despite differing in name, and a where-clause and a let expression can express the same computation in different syntactic forms. The model solution is the implementation of the programming task by an expert, expressing a valid strategy for solving the problem. Programming strategies are derived from these model solutions, where a strategy represents a formal description of the sequence of refinement steps a student may take when constructing their program. During model tracing, a student's normalized program is compared against intermediate programs described by these strategies to determine whether it still corresponds to a valid solution.

When a student's submission cannot be matched to model solution, Ask-Elle falls back on property-based testing. The program written by the student is tested against randomly generated test cases to determine whether it satisfies the prop-

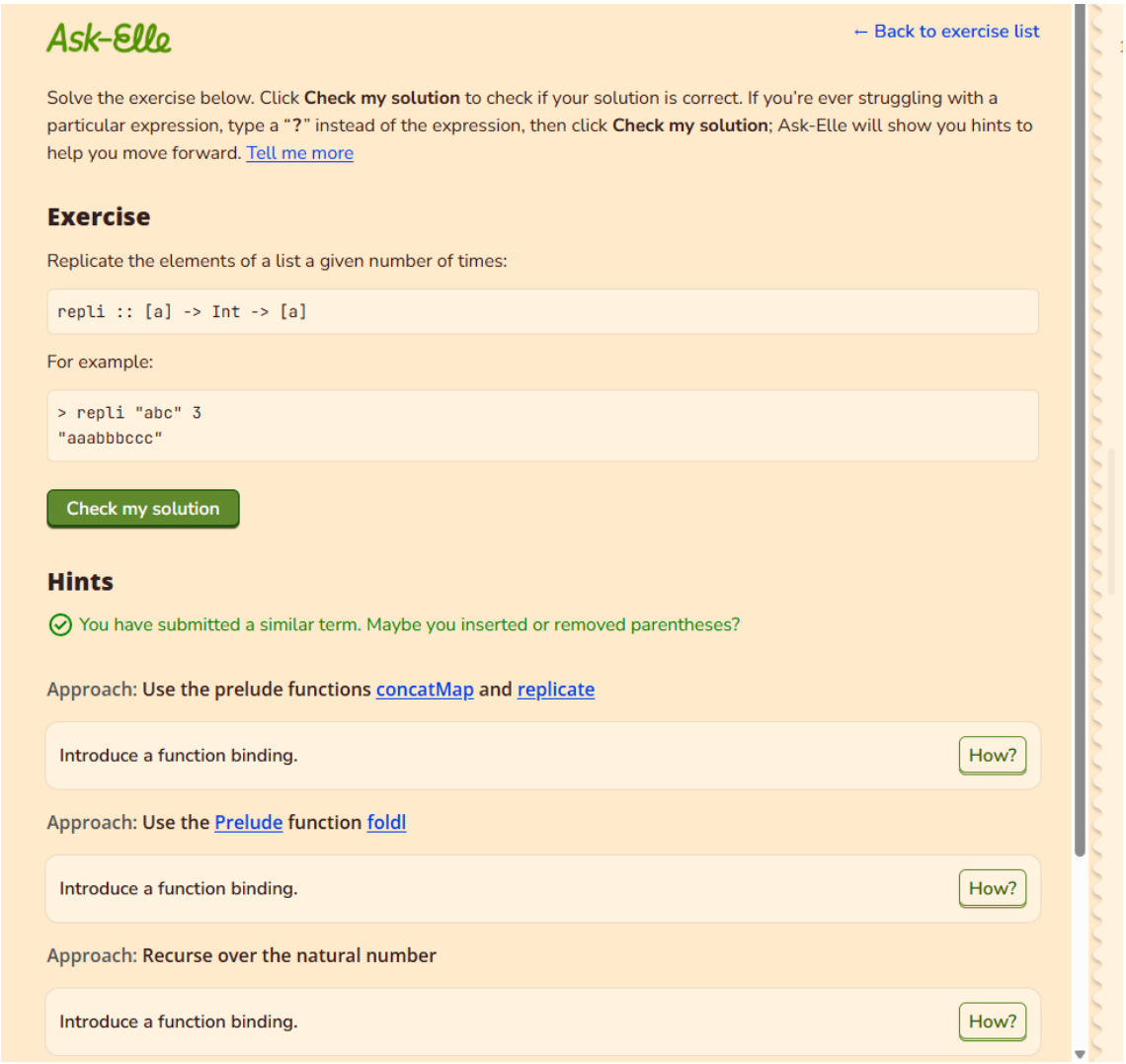

Figure 2.2: The functional programming tutor, Ask-Elle, revealing strategies the student can utilize in order to solve the exercise.

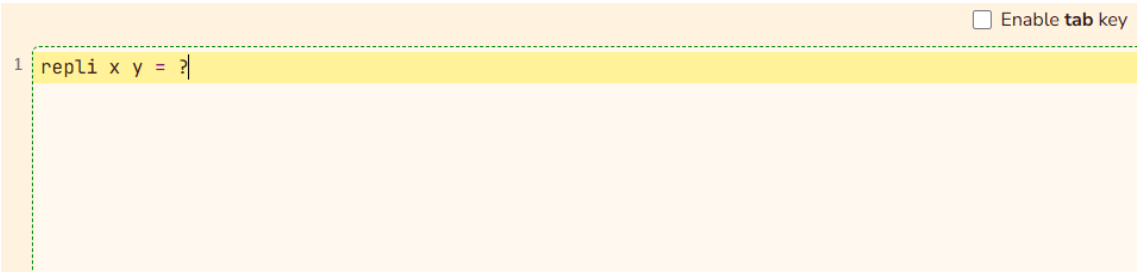

Figure 2.3: The editor of Ask-Elle in which the student uses to submit their code.

erties defined in the configuration file.

Submitted programs can be identified as belonging to one of four classifications [4]:

Compile error
Ask-Elle is unable to compile the program due to syntax or type errors that must be corrected by the student.

Matches model solution
Ask-Elle can match the submitted program with a model solution. Either the student is on the right track solving the exercise, or completely finished the exercise.

Counterexample
QuickCheck identifies a counterexample that falsifies a property, and feedback is provided to the student explaining why their program is incorrect.

Undecided
Ask-Elle neither successfully matches the submitted program to any model solution, nor finds a counterexample. Consequently, the program cannot be diagnosed as correct or incorrect. This classification is further divided into:

- **Tests passed:** All generated test cases succeed.
- **Discarded:** The majority of test cases are discarded, in almost all cases due to the program being undefined at too many places.

An example of a submission classified as Undecided is shown in Figure 2.4.

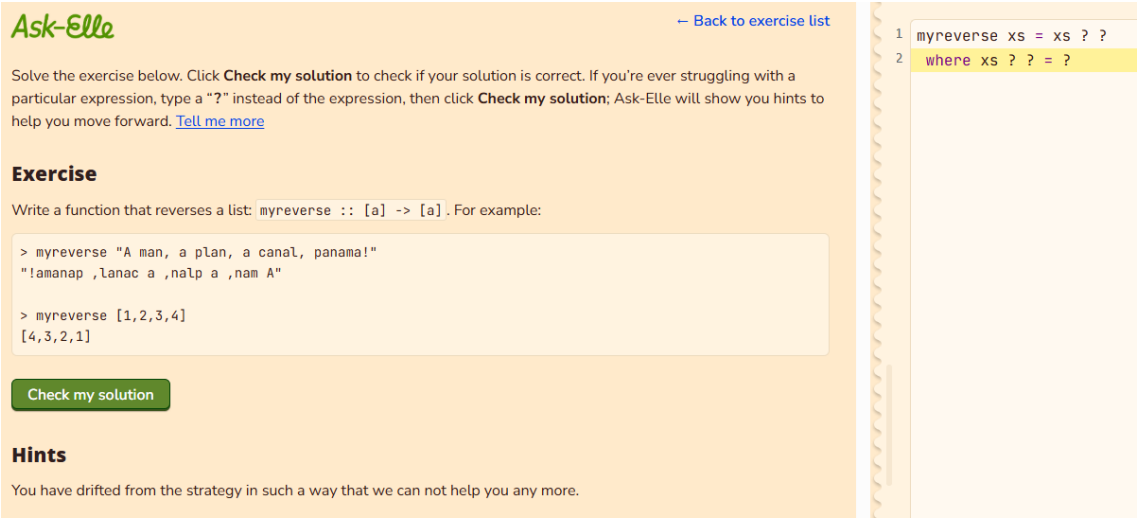

Figure 2.4: Ask-Elle providing limited feedback for a submission classified as Undecided.

3
Method

The core of this research involves extending the existing Ask-Elle architecture with an auxiliary GenAI component, as depicted in Figure 3.1. While Ask-Elle effectively handles Compile error, Matches model solution, and Counterexample classifications, it currently fails to provide meaningful pedagogical guidance for Undecided submissions. The proposed hybrid system bridges this gap by routing these scenarios through a dedicated GenAI component designed to process cases where standard reasoning cannot produce a definitive result. To manage the complex and version-dependent library requirements of Ask-Elle, a Docker-based Dev Container was employed to provide a standardized execution environment [24].

3.1 System Architecture

To generate a complete, testable code solution using GenAI, the student's submission is first processed by Ask-Elle's standard analysis pipeline. Only when Ask-Elle is unable to match the submission to a predefined model solution, cannot find a counterexample for the incomplete program, and too many test cases are discarded, does the GenAI component activate. To ensure long-term flexibility and maintainability, the GenAI component is implemented using a plugin architecture. By abstracting the API communication layer, the system can switch between different GenAIs.

The central mechanism of the architecture is a validation loop designed to ensure that GenAI-produced code is grounded in semi-formal verification using QuickCheck. Within this framework, Ask-Elle functions as a gatekeeper, maintaining system integrity, by subjecting AI-generated code to the same analysis pipeline (with the exception of model tracing) used for student submissions, which performs compilation and executes QuickCheck property tests.

The process is iterative, if the generated code fails at any stage of validation, the resulting error logs are passed back to the LLM as supplementary context. The LLM utilizes this feedback to refine and regenerate the solution, targeting the specific failures identified by the compiler or the property tests. Only after a generated solution successfully passes each validation stage is a hint generated by the LLM and provided to the student. The internal flow of this iterative process is depicted in Figure 3.2.

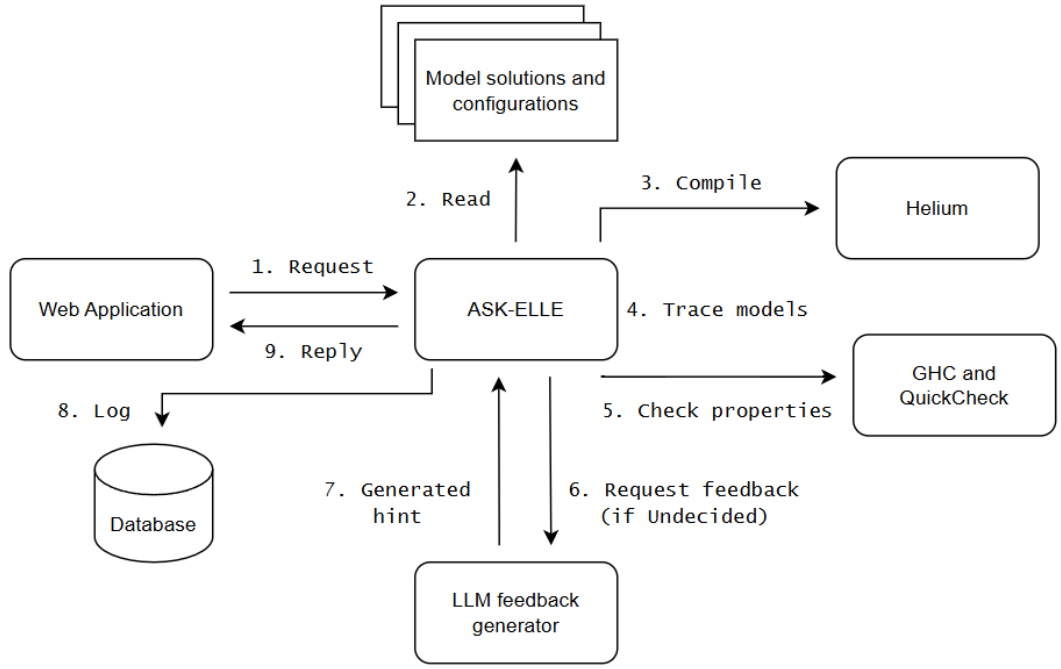

Figure 3.1: Component overview of the AI-integrated Ask-Elle system architecture.

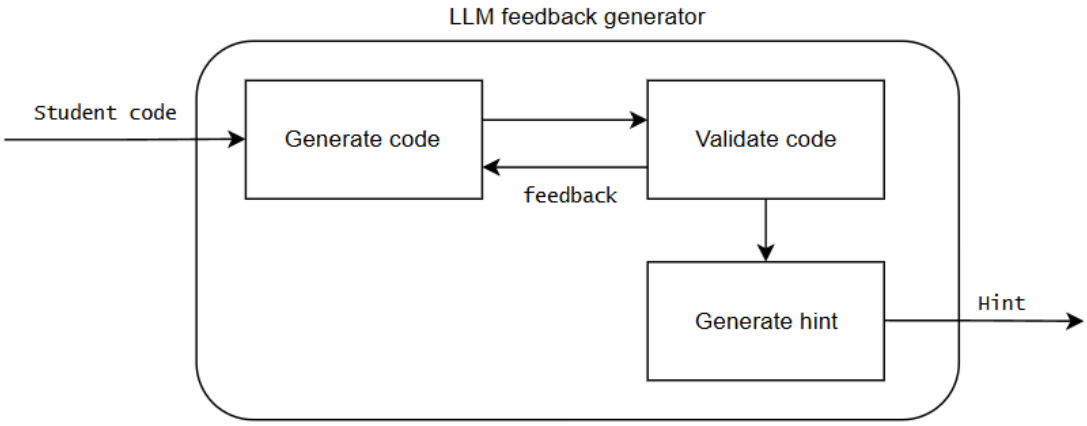

Figure 3.2: Internal iterative loop of the LLM feedback generator.

3.1.1 AI-Driven Remediation Workflow

Now that we have outlined the high-level architecture, we continue with detailed implementation of the workflow triggered by a Discarded submission. It covers the technical composition of the prompts, the specific instruction hierarchies customizing the LLM's behavior, and the operational constraints designed to maintain the educational effectiveness of the feedback. We show the data flow from the initial code-completion stage through the iterative verification loop, concluding with the generation of a pedagogical hint.

1. **Initial Code Generation**: When a Discarded classification is triggered, the system constructs a context-rich prompt containing the student's incomplete code, the exercise description, and the properties for that exercise. The inclusion of properties at this stage is a strategic design choice explained in Section 6.3. The prompt (see Appendix A.1) is designed to transform the LLM into an automated Haskell code-completion engine. The construction of this prompt is guided by a strict instruction hierarchy to ensure the LLM respects the student's original intent while working towards a verifiable solution. To facilitate automated extraction, the LLM was constrained to return all output exclusively within `<haskell_code>` tags.

    - **Priority 1 (Hole Filling)**: The LLM is primarily tasked with replacing holes represented by the `?` character with valid Haskell expressions.

    - **Priority 2 (Code Preservation)**: The LLM is instructed to keep the existing student code exactly as is, modifying only the holes where possible.

    - **Priority 3 (Minimal Repair)**: Only if a correct solution is logically impossible via hole-filling alone is the LLM permitted to perform minimal repairs on the surrounding code.

    - **Last Resort (Strategy Change)**: A fundamental change to the algorithm (e.g., switching from recursion to a fold) is only allowed if the student's current approach is fundamentally incapable of satisfying the exercise requirements.

2. **The Automated Verification Loop**: The solution returned by the LLM is automatically being tested and is treated as a new submission and must pass two criteria (the acceptance criteria):

    - **Compilation**: The code must compile without errors.
    - **Property Validation**: The code must satisfy all property-based tests defined for the exercise.

    Within this gatekeeping stage, the system distinguishes between several outcomes:

    `CompileError`: Triggered if the code fails the compilation stage.

    `Failed`: Triggered if the code compiles but QuickCheck identifies a counterexample.

    `RuntimeError`: Triggered if the validation process encounters an execution-level exception or a timeout during the testing phase. This status captures specific diagnostic failures, such as potential infinite loops (e.g., Diagnosing

timed out...") or type-level inconsistencies encountered during property checking (e.g., Testing your solution gave a type error..."). The latter can occur when QuickCheck must generate random values for the input types of a property. If the student's code contains overly general or ambiguous type signatures, Haskell's type system may not be able to determine what concrete type to instantiate when generating test inputs.

`Succeeded`: The only state that allows the pipeline to proceed to pedagogical hint generation.

Any result other than `Succeeded` (with the exception of `DontKnow`, which terminates the process) automatically triggers the iterative self-correction loop, where the specific error logs are fed back to the LLM as a new prompt (see Appendix A.2). This prevents any logically flawed or broken code from influencing the student's learning process. This verification loop repeats until a functionally correct solution is verified or the system reaches its predefined limit of three total attempts (which can be configured). If the system fails to produce a valid solution within these three attempts, the process terminates, and no guided hint is provided to the student for that submission.

3. **Pedagogical Hint**: Once the system's gatekeeper confirms a `Succeeded` status, indicating that a functionally correct completion has been verified, the workflow transitions from code generation to pedagogical guidance. This stage utilizes yet another prompt (see Appendix A.3) that explicitly redefines the LLM's role from an automated completion engine to a supportive Haskell tutor "whose sole objective is to provide a small nudge" toward the student's next intermediate step. The prompt provides the LLM with the student's original submission and the verified solution generated by the LLM.

   The LLM's task is to identify the single most important conceptual or logical fix required in the current submission. This task is carried out through a specialized prompt that casts the LLM as a supportive Haskell tutor, governed by strict constraints to ensure the feedback remains educationally effective:

   - **Singularity**: The LLM is tasked with providing exactly one hint. It is specifically instructed to avoid additive phrases like "also" or "additionally".

   - **No Direct Answers**: The LLM is strictly forbidden from providing code snippets or revealing any part of the reference solution.

   - **Formatting and Length**: To ensure the feedback is concise, the hint is limited to 2-3 sentences, must not use markdown, and must be phrased as a single guiding question or observation.

4. **Feedback Delivery**: The pedagogical hint is returned and presented to the

student via the web application interface.

The current prompt designs are the result of an iterative development process that identified and corrected several pedagogical and technical limitations in earlier versions. Specifically, the "Singularity" constraint was introduced because early iterations of the prompt frequently resulted in the LLM providing the entire solution path through the use of additive language. By explicitly prohibiting words such as "also" or "additionally", the system ensures the tutor remains focused on a single immediate step rather than providing the student with a comprehensive list of required changes.

Similarly, the enforcement of `<haskell_code>` tags was a critical adjustment to ensure the stability of the automated verification loop. Without these strict formatting boundaries, the LLM often provided conversational commentary, greetings, or explanations alongside the code. Since the `safeCompile` function in Ask-Elle attempts to process the LLM's response, any included natural language would be passed directly into the compilation stage, resulting in an automatic failure as it attempts to parse non-Haskell language as functional code.

To resolve this, the system utilizes a function called `sanitizeCode` to specifically locate and isolate the content within the `<haskell_code>` tags. This allows the pipeline to extract only the relevant code block while filtering out unnecessary conversational elements or markdown code. This design ensures the testing process remains focused solely on the generated solution, preventing non-code text from disrupting the verification pipeline.

3.2 Data Collection and Dataset

To evaluate the effectiveness of the hybrid pipeline, a structured approach was taken to select and process real-world student data. We detail the origin of the data and the preprocessing phase below.

3.2.1 Student Submissions

The primary data source for testing the prototype was a comprehensive database containing historical student submissions from 2014 and 2015. This database logs code attempts made by students using the Ask-Elle system. Initially, the filtering process identified 86 data points that had been classified as "Discarded" by the legacy version of Ask-Elle used during the original collection period. However, the current version of the system successfully processed many of these historical submissions without requiring generative assistance. Consequently, only 36 of these 86 data points were classified as Discarded by the current version and carried forward into the testing phase.

The remaining 36 submissions were provided as input to a benchmarking tool developed for this study (as detailed in Section 3.3) to verify the system's performance.

This tool assessed whether the AI, operating as a code-completion engine, could successfully populate code holes to generate a functionally valid, testable solution along with a hint.

Selection of Exercises
The experiments were conducted on 15 functional programming exercises in Haskell, designed to represent a spectrum of common programming concepts and difficulty levels. The 36 total student submissions were distributed across these 15 exercises, with the volume of data points varying by exercise. For instance, while some exercises had few data points, the "repli" exercise (replicating elements) accounted for 5 of the 36 total submissions. The distribution of discarded submissions per exercise is shown in Figure 3.3.

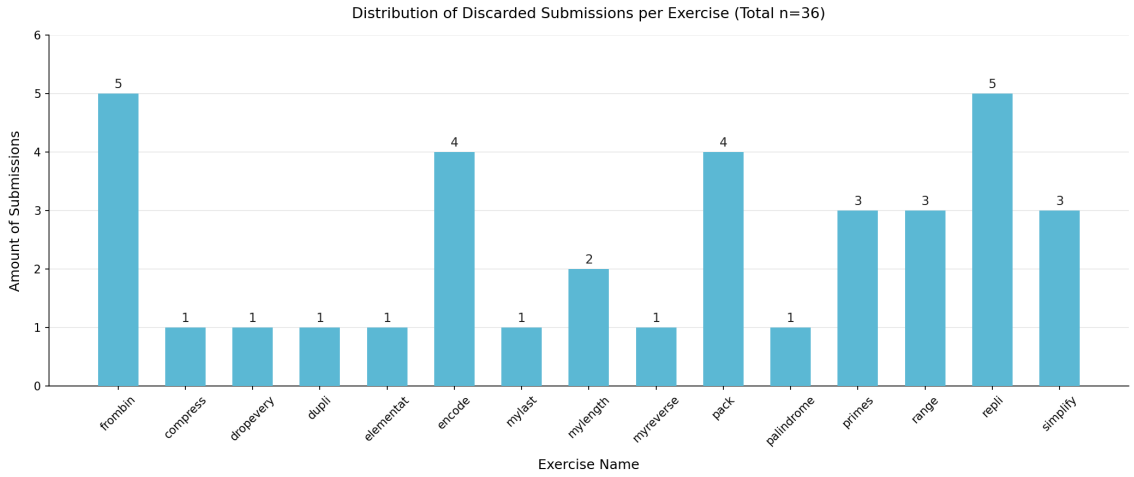

Figure 3.3: Distribution of discarded submissions per exercise.

3.3 Benchmarking and LLM Selection

The effectiveness of the pipeline is determined by its ability to reliably transform Discarded student submissions into constructive learning opportunities. Because the gatekeeper must verify all AI-generated code, the system must balance technical correctness with the low latency expected in a tutoring environment. During the development phase, initial experimentation was conducted using local LLMs via Ollama [25]. However, it quickly became evident that these LLMs lacked the reasoning capabilities required for the GenAI pipeline. They frequently failed to generate code that could pass either the compilation or testing stages, and the resulting hints were found to be pedagogically insufficient. Consequently, the evaluation shifted to more robust alternatives where two LLMs were evaluated, OpenAI's GPT-5.3-Codex and Claude Sonnet 4.6. The selection of the LLM was based on an evaluation of three performance metrics:

- Correctness: The success rate of the LLM in producing code that satisfies the requirements of both the compilation and test stages.

- Response time: How quickly the AI produces feedback.
- Pedagogical quality: How instructive the feedback is for learning. (Assessed subjectively by ourselves)

To facilitate a data-driven LLM selection, we developed two specialized tools. The first is a benchmarking tool that enables comparative analysis of LLM performance. The second tool generates the necessary data by running all student submissions against each LLM across every prompt configuration (as listed below), gathering the generated code, pedagogical hints, remediation times, and validation results for each combination. Together, these tools provide the empirical foundation necessary to address RQ1, RQ2, and the technical aspects of RQ4 regarding code-generation reliability.

- Baseline: The LLM received only the student's incomplete code and the exercise description.
- With Model Solution, Without Property Tests: In addition to the baseline, the LLM was provided with model solutions.
- Without Model Solution, With Property Tests: In addition to the baseline, the prompt included formal QuickCheck properties.
- With Model Solution and Property Tests: The most context-rich configuration, extending the baseline with both model solutions and QuickCheck properties.

This approach allowed the team to quantitatively verify which configuration yields a higher rate of correct solutions.

While technical benchmarking verified the efficiency and accuracy of generating correct code of the pipeline, the final stage of the methodology shifted focus toward the pedagogical value of the feedback. This phase addresses the pedagogical aspects of RQ4 and the broader impact on student learning explored in RQ3.

3.3.1 Blind A/B Evaluation

To answer RQ4 and assess the pedagogical quality of the feedback, we supplemented the quantitative metrics with a blind A/B evaluation of the generated hints. This evaluation allowed us to detect hallucinations while completing the dataset required for final LLM and prompt configuration selection. A self-developed web application displayed two hints side-by-side, masking the generating LLM and prompt configuration. For each session, the tool randomly paired hints derived from the same student submission and exercise to directly compare different configurations.

We manually reviewed each pair to select a preferred hint based on three pedagogical requirements: whether the feedback directly builds upon the student's existing code structure instead of forcing an algorithmic shift, whether it guides the student toward the underlying logic rather than outright revealing the next mechanical step, and whether the text uses clear language free of technical hallucinations. If the hints were of equal quality (whether equally helpful or equally flawed), the trial was recorded as a tie.

All ratings were stored in a database and aggregated into a leaderboard. Final rankings were determined by the decisive win rate ($W/(W + L)$), excluding ties from the denominator. Any identified hallucinations were separately documented during this process.

3.3.2 LLM Scaling Analysis

To further explore the economic and operational feasibility of the feedback pipeline, we conducted a comparative analysis using GPT-4.1-nano, a significantly smaller and more resource-efficient LLM. The primary objective of this sub-study was to determine if a smaller LLM could produce results comparable to larger, more resource-intensive architectures, such as GPT-5.3-Codex or Claude Sonnet 4.6, when provided with the exact same prompts and configurations. Specifically, this analysis investigated whether the reasoning limitations of a more efficient LLM could be mitigated by the contextual depth of the prompts, testing if it could achieve similar performance benchmarks when utilizing the same context as the high-tier LLMs.

3.4 User Study and Pedagogical Evaluation

Finally, a user study was conducted to address RQ3 by evaluating the pedagogical impact of the generated hints on actual students. The primary objective was to assess whether the integration of GenAI with Ask-Elle improves the quality and educational value of feedback compared to the standalone system. Specifically, the study examined the instructional efficacy of these hints, determining if they could effectively guide students without providing excessive assistance that might reduce the exercise's educational value [26]. The exercises assigned to the participants and the structure of the qualitative survey are detailed in Table 3.1 and Table 3.2, respectively.

Exercise	Type Signature & Description	Example Output
`myreverse`	`[a] -> [a]` Reverses list elements.	`[1,2,3,4]` → `[4,3,2,1]`
`repli`	`[a] -> Int -> [a]` Replicates elements n times.	`"abc" 3` → `"aaabbbccc"`
`split`	`[a] -> Int -> ([a], [a])` Splits list at given index.	`[1..5] 3` → `([1,2,3],[4,5])`
`dropevery`	`[a] -> Int -> [a]` Drops every n-th element.	`[0..9] 3` → `[0,1,3,4,6,7,9]`

Table 3.1: Haskell exercises for the user study.

Category	Qualitative question
Participant Survey	
Clarity	What terminology was confusing?
Utility	Was the hint too vague or detailed?
Logic	Did it fix the "why" or the "what"?
Scaffolding	Did it respect your original intent?
Singularity	Were you overwhelmed by information?
Timing	Did the wait break your flow?
Engagement	Did the tool provide sufficient assistance?

Table 3.2: Participant Survey

4
Results

The answers for the RQ1, RQ2 and technical aspects of RQ4 regarding code-generation reliability are based on the data captured by the benchmarking tool. This tool provided data on success rates and remediation latencies, allowing evaluation of how effectively the GenAI integration complemented Ask-Elle's rule-based mechanisms. To fully answer RQ4, we reviewed a sample of generated hints to detect any potential hallucinations. Complementing this technical data, the user study served as the primary source of evidence for answering RQ3. By observing 5 participants as they interacted with the AI-integrated Ask-Elle environment, the study captured qualitative feedback on improvement of the quality and educational value of feedback.

4.1 Time Distribution

Significant delays between a student's submission and the system's response disrupt the interactive user experience. Therefore, measuring the pipeline latency is necessary to determine whether the pedagogical value added by the LLMs justifies their computational overhead.

The data presented in Figures 4.1 and 4.2 illustrates the total remediation latency. This metric tracks the time from the moment the system determines that AI assistance is required until the pipeline reaches a terminal state: either the successful generation of a hint, the exhaustion of the three-attempt limit, or a premature termination due to a system error.

Because of the exclusion of the initial Ask Elle diagnosis, the latency presented in the following graphs will not be representative of the total end-to-end response time experienced by the user, but rather the net duration of the AI-driven remediation loop.

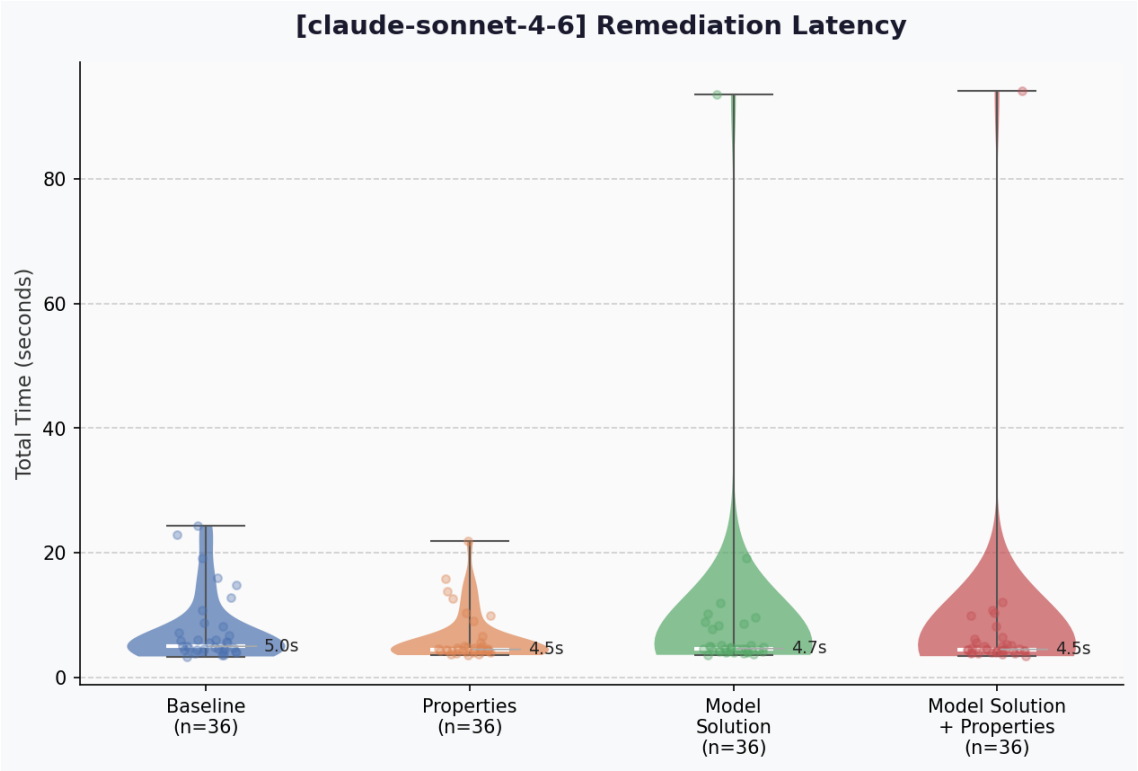

Figure 4.1: Claude Sonnet 4.6 Remediation Latency.

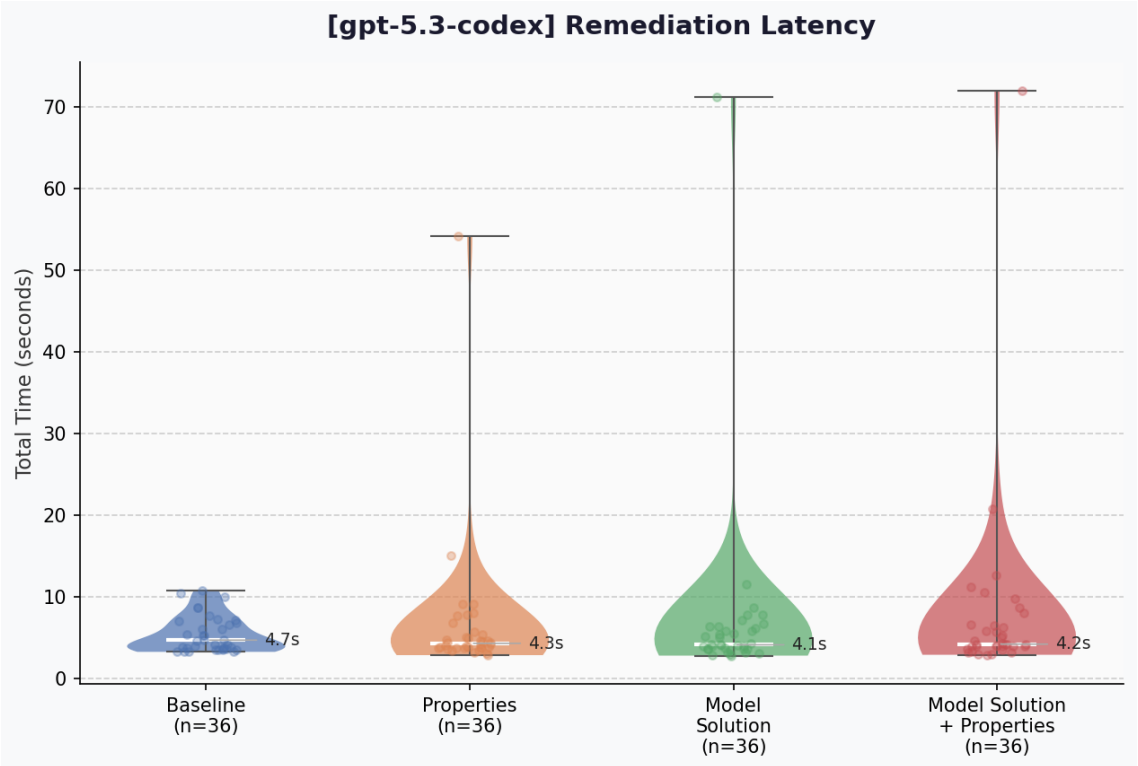

Figure 4.2: GPT-5.3 Remediation Latency.

A comparative analysis of the distributions reveals that GPT-5.3-Codex consistently

maintains a lower median latency across all test cohorts, ranging from 4.1s to 4.7s, compared to Claude Sonnet 4.6's range of 4.5s to 5.1s. While the "Baseline" and "Properties" configurations show relatively tight clusters for both LLMs, the introduction of "Model Solution" data significantly increases the variance.

Claude Sonnet 4.6 exhibited extreme outliers with response times exceeding 90s, whereas GPT-5.3-Codex's maximum latency remained lower, peaking at approximately 72s. These specific outliers occurred in instances where the LLMs utilized all three attempts before reaching a terminal state.

4.2 Status Distribution

Frequent failures in generating correct code prevents the pipeline from returning a hint, directly reducing the system's availability when a student requests help. Measuring the distribution of terminal execution statuses is therefore necessary to determine whether the prompt configurations yield a high enough success rate to be dependable.

The distributions illustrated in Figure 4.3 and Figure 4.4 represent the status of each exercise attempt across the four prompt configurations. It is important to note that these metrics do not reflect the specific number of attempts required to reach a solution. Instead, a "Success" status indicates that the LLM successfully generated functionally correct code within the predefined three-attempt limit. Conversely, non-success statuses (Failed, CompileError and RuntimeError) indicate that the system used all three allowed attempts but was unable to find a successful result.

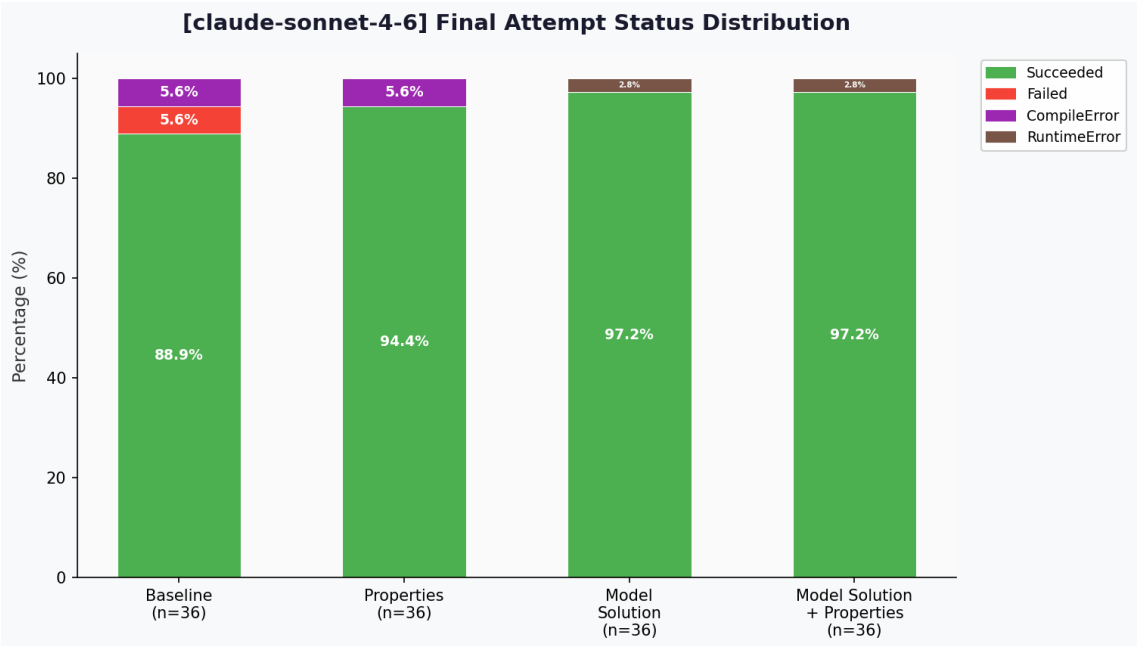

Figure 4.3: Claude Sonnet 4.6 Final Attempt Status Distribution

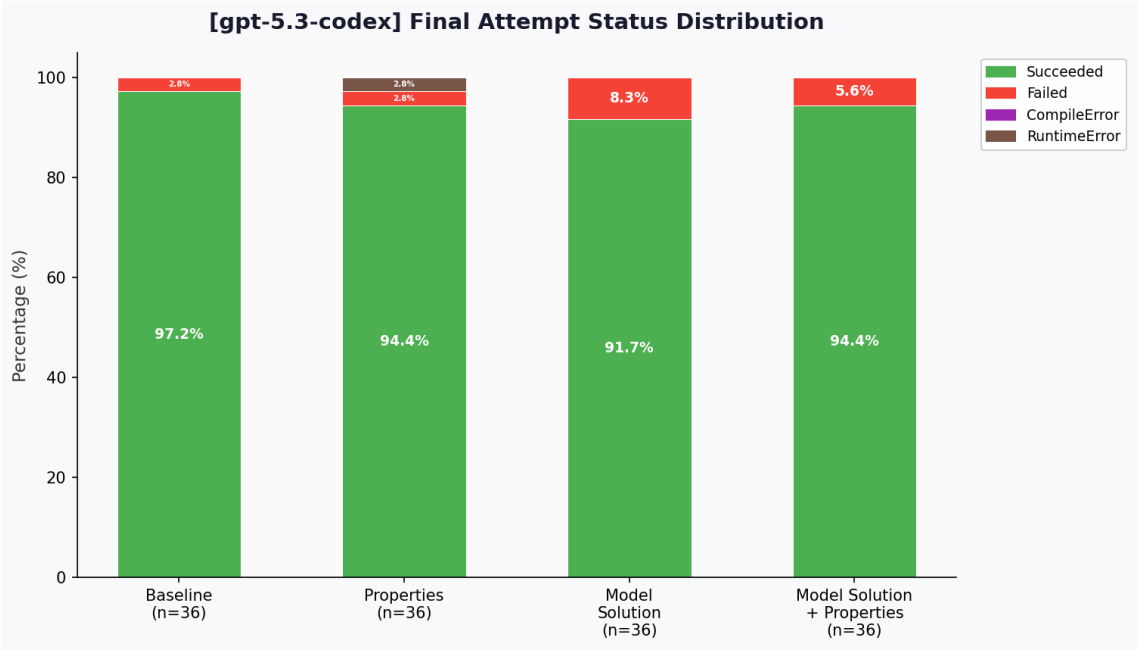

Figure 4.4: GPT-5.3-Codex Final Attempt Status Distribution

The graph for GPT-5.3-Codex illustrates divergent performance trajectories across the four evaluated prompt configurations. GPT-5.3-Codex demonstrated a strong initial Baseline success rate of 97.2%, but experienced a fluctuating regression as context increased, dropping to 94.4% with Properties, reaching a low of 91.7% with the Model Solution, and ending at 94.4% for the combined Model Solution + Properties configuration. Conversely, Claude Sonnet 4.6 follows an improvement curve. Starting from an 88.9% Baseline, the LLM's performance rose to 94.4% with Properties and peaked at 97.2% in both configurations utilizing the Model Solution.

The observation that success rates for the majority of LLMs and prompt configurations consistently reached 94% or higher provides empirical evidence that the hybrid integration strategy detailed in Section 3.1 is effective. This finding directly addresses RQ1 by demonstrating that GenAI can be successfully harmonized with rule-based reasoning to address Discarded submissions that were previously beyond the system's reach.

Regarding the technical reliability aspects of RQ4, the consistently high success rates indicate that the prototype is dependable in generating code that satisfies all defined acceptance criteria. The system proves it can handle the nuances of functional programming with high precision.

4.3 Success Rate by Attempt

We analyze the efficiency of the feedback pipeline by examining how many attempts each LLM required to reach a successful state. While the previous charts showed the final outcomes, these heatmaps reveal the iterative performance, essentially showing

if the LLM gets it right immediately or if it needs to "learn" from the system's feedback to correct its mistakes. The success rate heatmaps (Figure 4.5 and Figure 4.6) illustrate the distribution of successful outcomes across the three-attempt limit. This metric helps determine whether the latency introduced by multiple regenerations is justified by an improvement in the LLM's ability to reach a verified solution.

To interpret these heatmaps correctly, it should be noted that the percentages are absolute and not relative. This means every percentage is calculated based on the total number of submissions (n = 36). For instance, if 90% of the submissions are solved in the first attempt, the remaining 10 percentage points move on to the second attempt. If the second attempt shows a success rate of 5%, this represents 5% of the original total ($n = 36$). This calculation method ensures that the sum of success percentages across all three attempts equals the total success rate shown in the earlier distribution graphs. However, these totals may not be exactly the same as the sum of success percentages in every case due to rounding errors. Because each individual attempt is rounded to one decimal place to make the heatmaps easier to read, the sum of those rounded numbers might differ from the total in the other graphs by a very small amount (usually 0.1%). These tiny differences are a result of the rounding process and do not indicate an error in the underlying data.

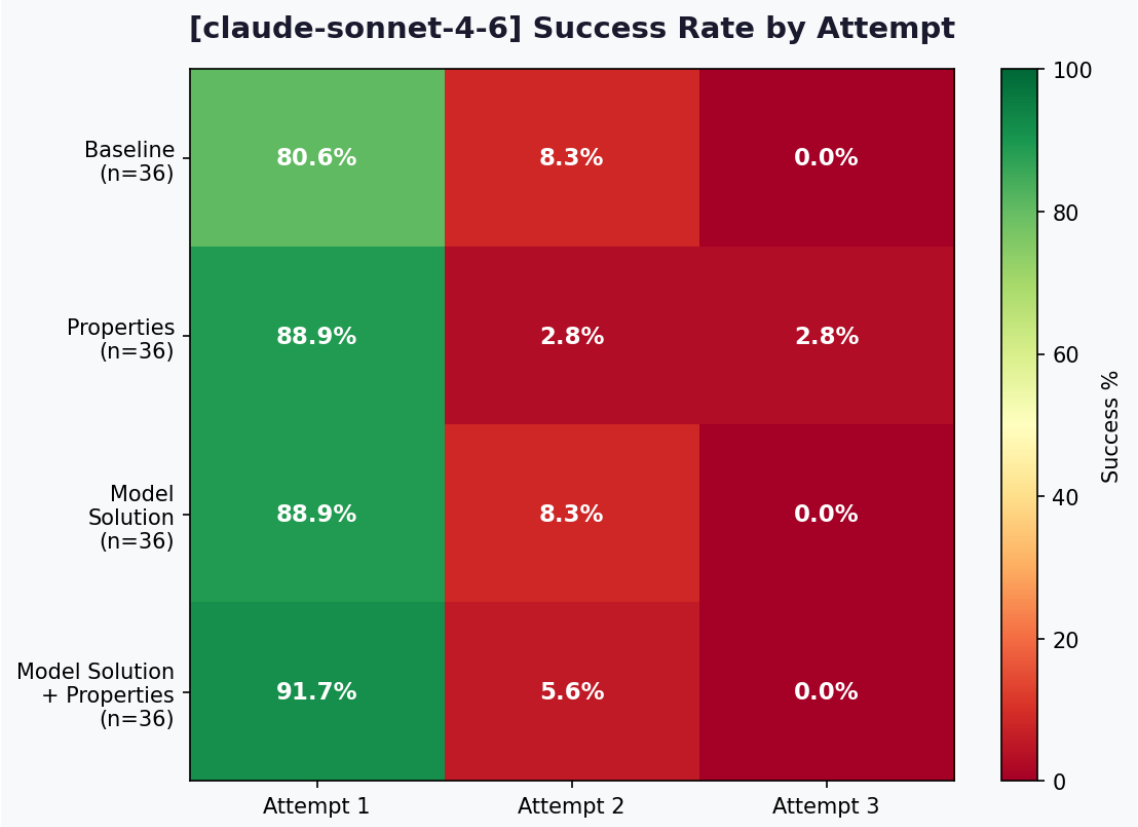

Figure 4.5: Claude Sonnet 4.6 Success Rate by Attempt.

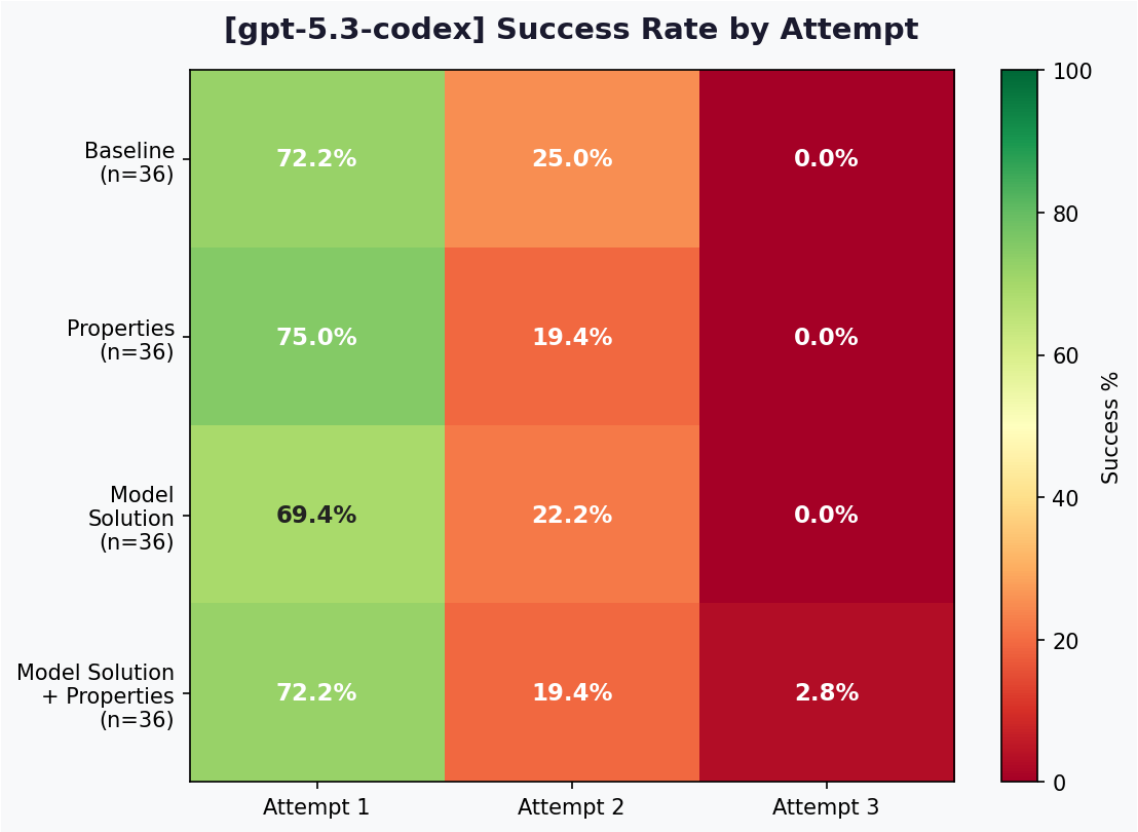

Figure 4.6: GPT-5.3-Codex Success Rate by Attempt.

Claude Sonnet 4.6 demonstrates high first-attempt precision across the configurations. However, the heatmaps also reveal a significant lack of iterative recovery. The LLM demonstrates a limited capacity for self-correction. After an initial failure, success rates drop significantly in later attempts, frequently reaching a terminal rate of 0.0%.

In contrast, GPT-5.3-Codex displays a lower success rate on the initial attempt across all configurations, with first-try success rates ranging between 69.4% and 75.0%. However, the LLM proves to be significantly better within the iterative loop. While fewer problems are solved immediately, a substantial percentage of cases (ranging from 19.4% to 25.0%) are successfully resolved during the second attempt. This indicates an enhanced capacity for self-correction, as it effectively leverages system-provided feedback, to fix its own logic in the following attempts.

This data offers a practical demonstration of the validation loop architecture explored in RQ2, where Ask-Elle's existing tools act as a gatekeeper. The fact that a significant portion of cases are successfully resolved in the second and third attempts, particularly by GPT-5.3-Codex, confirms that the system's ability to catch errors and feed them back to the LLM functions as intended. Instead of allowing a faulty first attempt to translate into a hallucinated hint, the tools, such as the Helium compiler and QuickCheck properties, effectively block incorrect code and provide the necessary diagnostic context for the LLM to refine its logic. This recov-

ery process serves as empirical evidence for the reliability of the validation pipeline.

4.4 Success Rate Per Exercise

While the aggregate success rates provide a general overview, the per-exercise heatmaps (Figure 4.7 and Figure 4.8) offer a more granular look at how prompt configurations affected specific tasks.

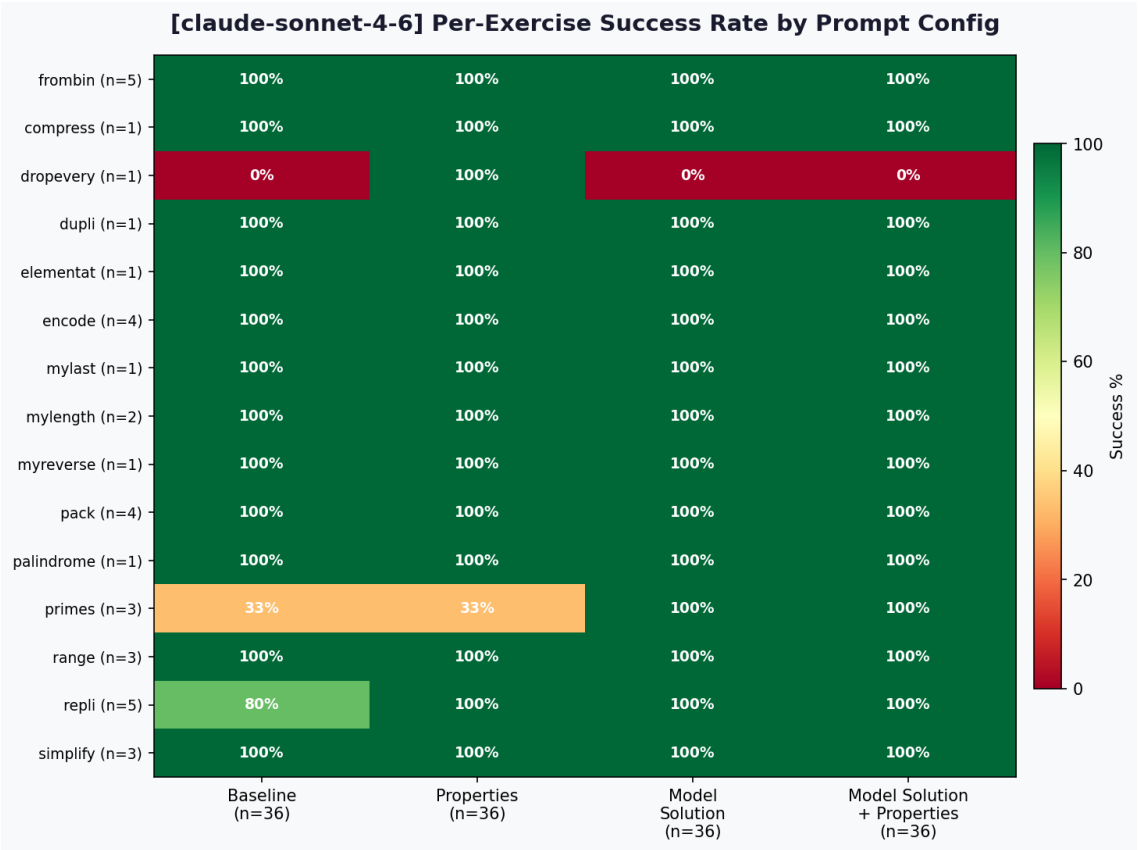

Figure 4.7: Claude Sonnet 4.6 Per-Exercise Success Rate.

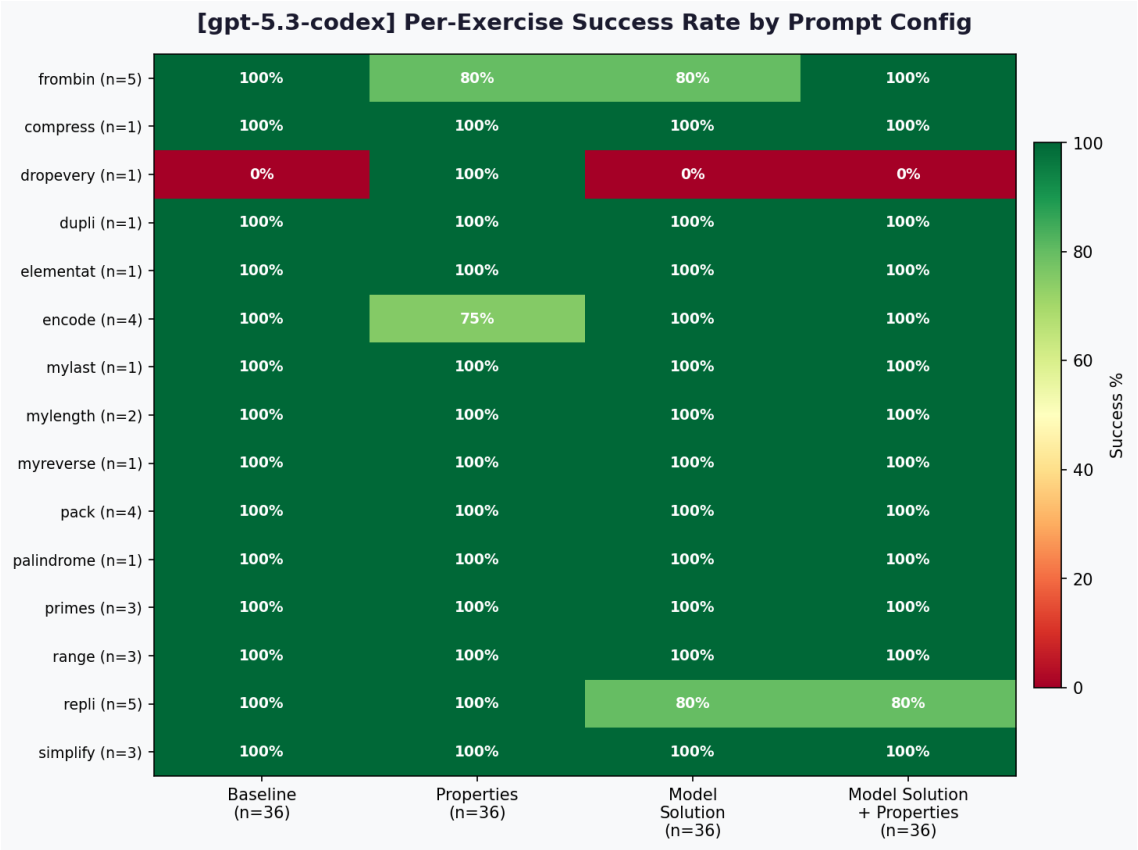

Figure 4.8: GPT-5.3-Codex Per-Exercise Success Rate.

These visualizations reveal that most exercises were solved with 100% consistency. While a small number of submissions remained unresolved, the `dropevery` exercise failed to yield a successful outcome across nearly all configurations. A detailed inspection of the `dropevery` exercise revealed that the exercise itself contained a critical defect, which may have caused the low success rate for the different configurations.

4.4.1 Defected dropevery Exercise

Within the Ask-Elle framework, model solutions are defined as valid solutions for every exercise. These implementations are assumed to be correct and are expected to satisfy all formal properties specified for the task. Consequently, any student submission that matches a model solution after normalization is considered correct and is exempt from further property-based testing.

In this instance, the student's code failed to match a model solution and was categorized as Discarded, which triggered the AI remediation workflow. The workflow produced a solution for the `dropevery` exercise (Table 3.1) that was semantically identical to the model solution.

Comparison of dropevery Implementations

Model Solution:

```
dropevery [] _ = []
dropevery list count =
    take (count - 1) list ++ dropevery (drop count list) count
```

AI-Generated Code:

```
dropevery [] _ = []
dropevery (x : xs) n =
    take (n - 1) (x : xs) ++ dropevery (drop n (x : xs)) n
```

Figure 4.9: Comparison of the model solution and AI-generated code for the dropevery exercise.

Unlike standard student submissions, AI-generated code is not normalized to facilitate model tracing, as discussed in Sections 3.1. Instead, it is subjected directly to the QuickCheck property-based test suite immediately after it is compiled. During this testing phase, QuickCheck generated randomized inputs, including non-positive integers ($n \leq 0$) for the second argument. This rigorous testing exposed a logic defect present in the model solution itself.

When $n \leq 0$, the expression `drop n xs`, returns the original list unchanged. The core issue lies in how the function handles the integer argument n, when it is a non-positive integer. The intended logic is to take $n - 1$ elements and then recurse on the remainder of the list after dropping n elements. As a result, the recursive call `dropevery (drop n xs) n` is invoked with identical arguments to the current call, producing infinite recursion. Neither the model solution nor the AI-generated solution define any guard or base case to handle non-positive values of n, meaning this behavior is present in both implementations.

This discovery reveals that the model solution for the `dropevery` exercise is incomplete, as it fails to account for non-positive values of the n parameter. While such values may fall outside the intended pedagogical use case, they remain entirely valid inputs from the perspective of the Haskell type system. As model solutions form the ground truth against which all student submissions are evaluated, a defective model solution undermines the integrity of the exercise. Specifically, if a student's submission matches a model solution after normalization, the system considers it correct and skips further testing. If that model solution contains a bug, such as the infinite recursion found in the `dropevery` exercise, the system will erroneously approve student code that contains that exact same bug. This inadvertently reinforces a technical misconception rather than correcting it.

Every prompt configuration that included the model solution as context at-

tempted to replicate the faulty implementation shown above in its first attempt. In contrast, configurations that excluded the model solution, specifically those provided with QuickCheck properties, were not biased by the defective model solution. This allowed the properties configurations for both LLMs to derive a functionally valid solution, whereas the baseline configurations proved unable to solve the exercise independently.

4.5 Normalized Edit Distance

The Levenshtein distance, or standard edit distance (ED), is defined as the minimal number of insertions, deletions, or substitutions required to transform one string into another [27]. However, ED is an absolute measure and does not account for the length of the strings being compared. In many applications, three edits in a four-letter word are far more significant than three edits in a hundred-letter word. To address the limitations of absolute distance, we normalize the distance by dividing by the max length of either string.

The normalized edit distance (NED) provides insight into the extent to which the LLMs rewrite the student's original code. A value closer to 0 indicates minimal syntactic changes, whereas a value approaching 1 suggests a complete algorithmic rewrite. The following figures illustrate the NED distributions across different prompt configurations for Claude Sonnet 4.6 and GPT-5.3-Codex, establishing a baseline for how heavily each LLM intervenes.

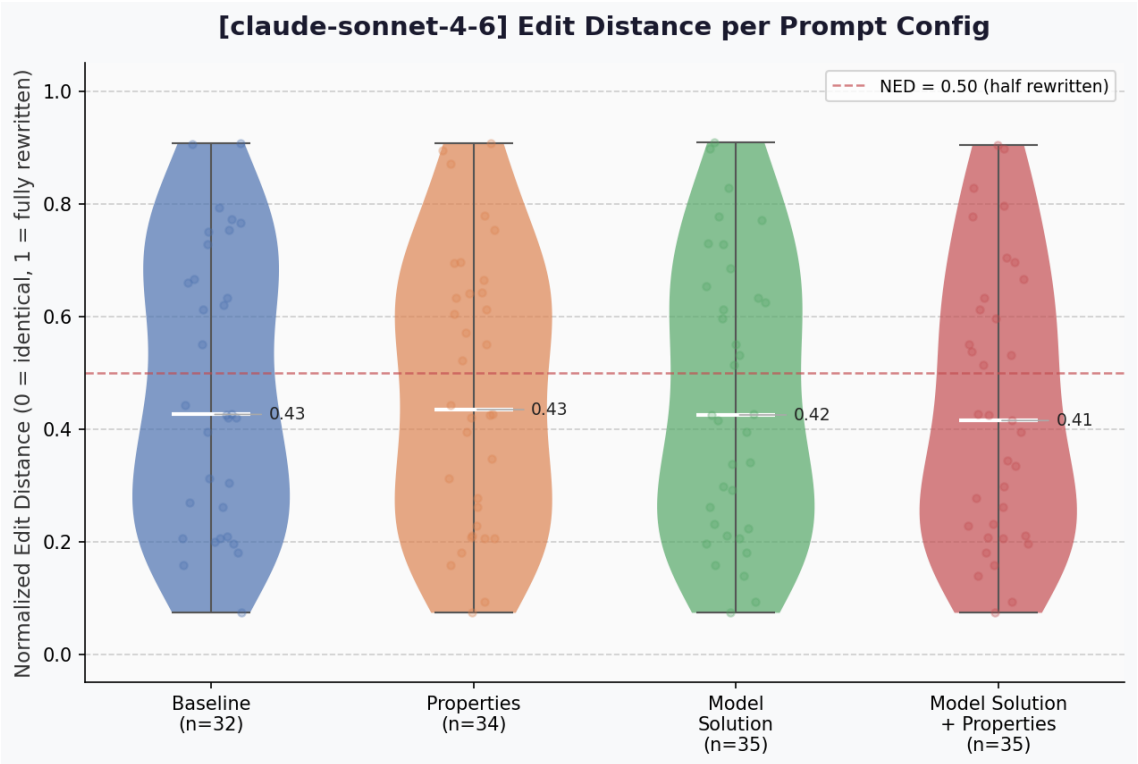

Figure 4.10: Claude Sonnet 4.6 Normalized edit distance.

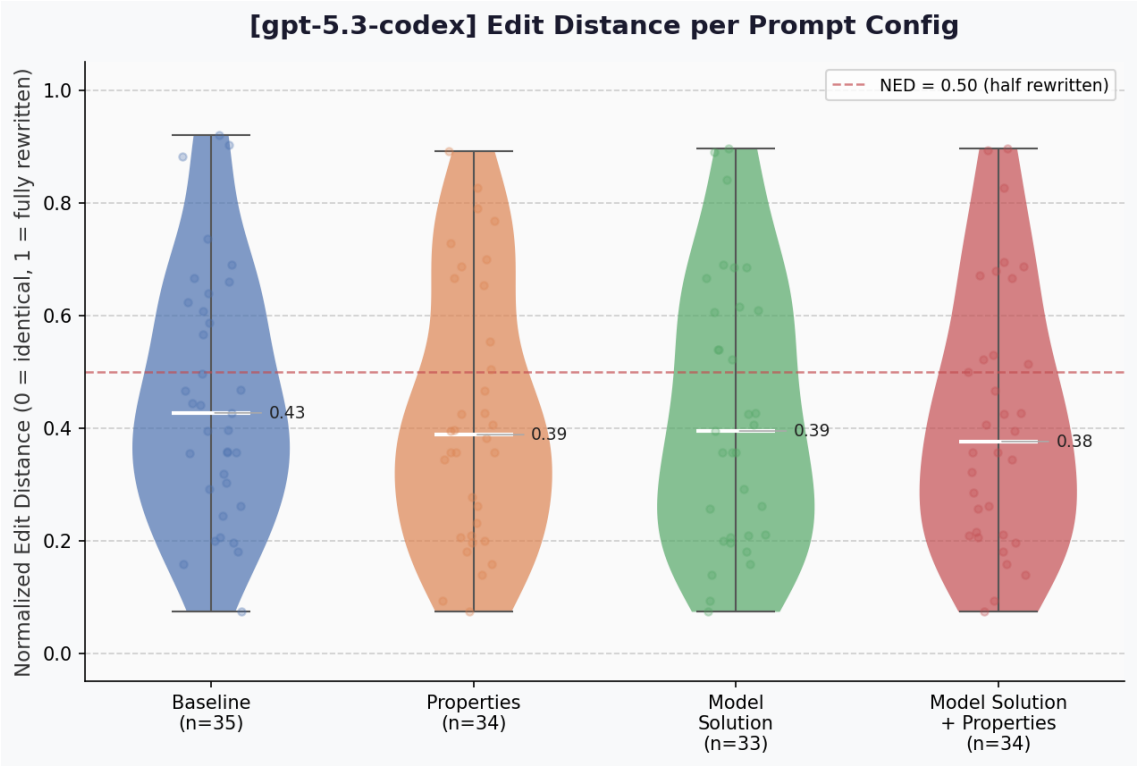

Figure 4.11: GPT-5.3-Codex Normalized edit distance.

As illustrated in Figure 4.10, Claude Sonnet 4.6 maintains a relatively stable intervention level across all prompt configurations, with all median NED values clustered between 0.41 and 0.43.

In contrast, GPT-5.3-Codex (Figure 4.11) shows a reduction in NED when additional context is provided. While its Baseline NED is the configuration with the highest median, the LLM becomes more conservative in all other configurations. Both the Properties and Model Solution variants yield a median NED of 0.39, reaching a minimum of 0.38 when both properties and model solutions are included in the prompt.

4.6 LLM scaling

Larger models, while more capable, come at a significantly higher financial cost per token and may introduce greater latency. GPT-5.3-Codex for instance is priced at $1.75 per million input tokens and $14.00 per million output tokens, whereas GPT-4.1-nano is priced at $0.10 and $0.40 respectively. In a tutoring environment where the system may process a high volume of student submissions simultaneously, this cost difference becomes operationally significant. It is therefore worth investigating whether a smaller, more resource-efficient LLM could achieve comparable performance.

The benchmarking data for GPT-4.1-nano indicates that the inclusion of

model solutions is a critical dependency for smaller architectures, as it significantly enhances the LLM's ability to reach a successful terminal state. However, the success is largely driven by the LLM's tendency to disregard the student's original logic, often opting to generate an entirely independent solution rather than building upon the student's existing code. This lack of intent preservation is reflected in a significantly higher edit distance compared to the larger models.

Furthermore, GPT-4.1-nano exhibited a markedly lower first-attempt success rate, necessitating a higher number of iterative cycles within the validation loop to produce a functionally correct result. The relationship between remediation latency and success rates per attempt is further elaborated in Section 6.2.

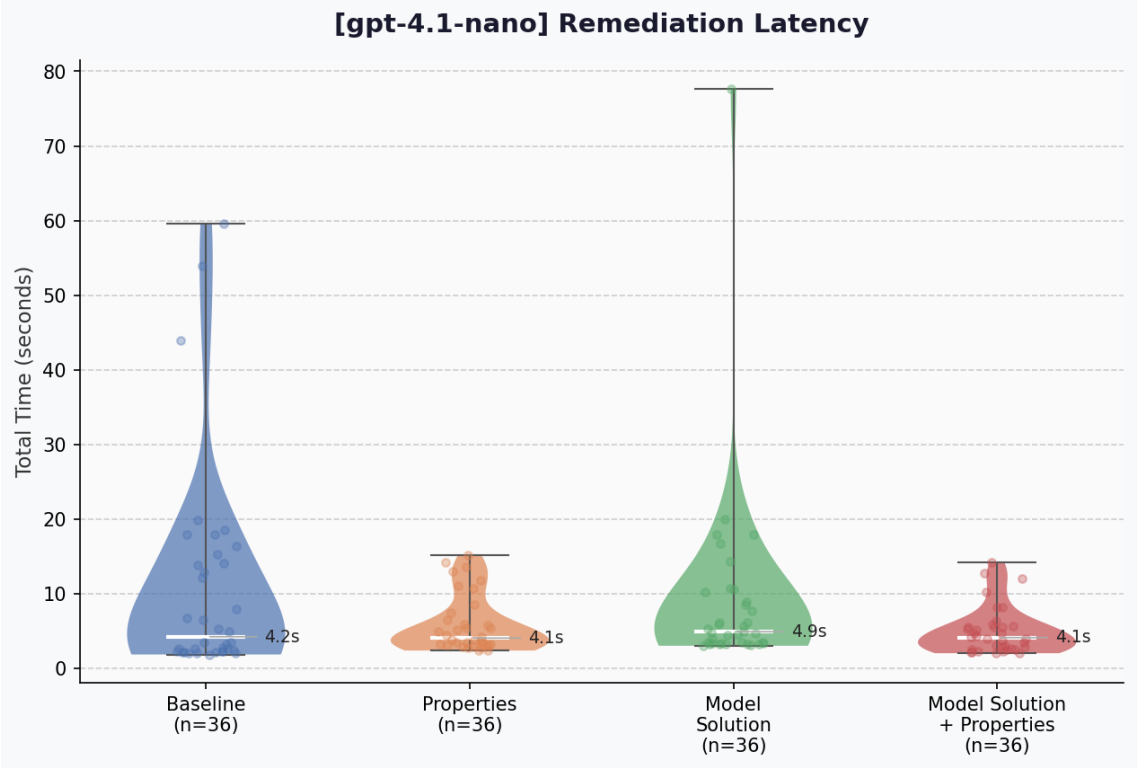

Figure 4.12: GPT-4.1-nano Remediation Latency.

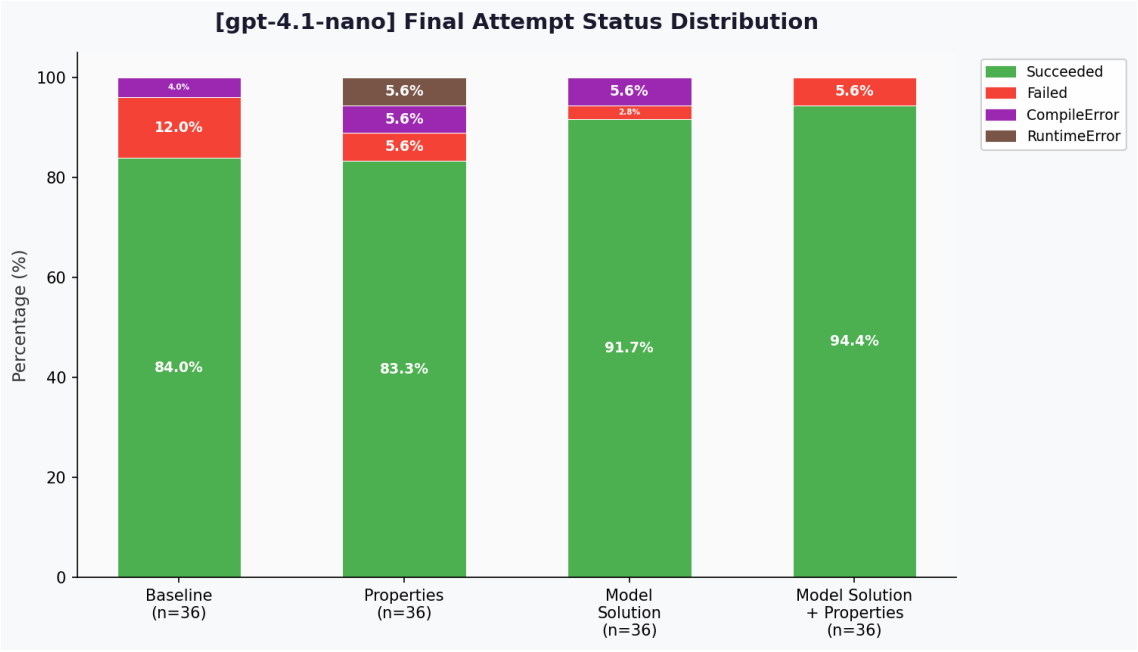

Figure 4.13: GPT-4.1-nano Final Attempt Status Distribution.

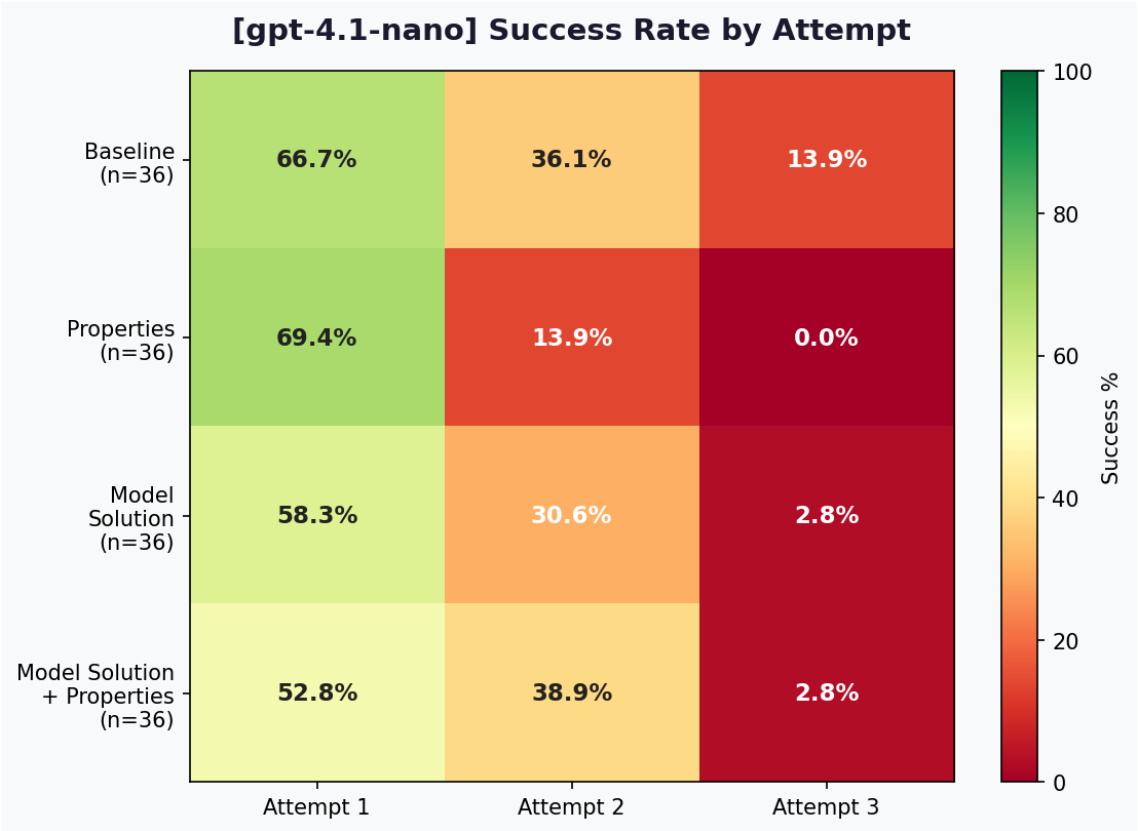

Figure 4.14: GPT-4.1-nano Success Rate by Attempt.

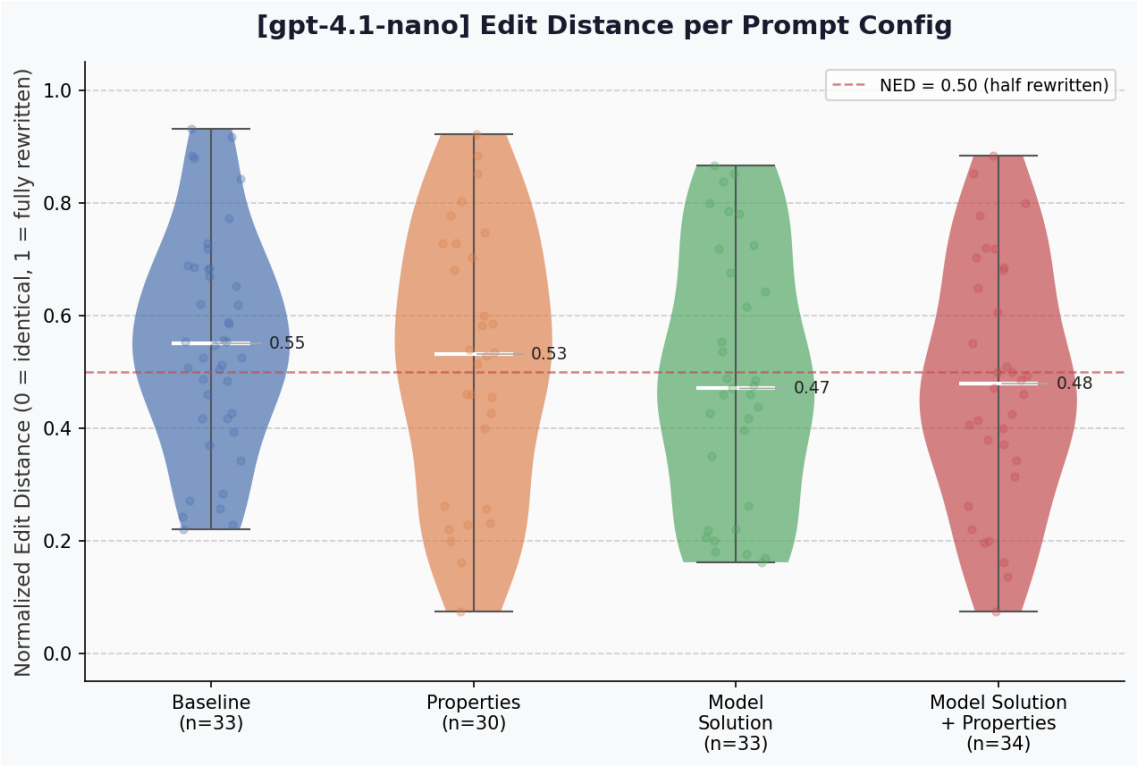

Figure 4.15: GPT-4.1-nano Normalized edit distance.

4.6.1 Validating Hints

To evaluate pedagogical effectiveness, 62 A/B trials were conducted comparing randomly selected hint variants. In each trial, two hints generated by different combinations of LLM and prompt configurations for the same student submission were displayed side-by-side, and the preferred hint was selected based on the pedagogical criteria described in Section 3.3.1. The resulting leaderboard reveals that the GPT-5.3-Codex properties configuration emerged as the most successful variant, recording 8 wins, 2 losses, and 6 ties (80% win rate when excluding ties). Ties generally occurred when both configurations produced hints of indistinguishable pedagogical value. Testing was concluded at 62 trials because the blind evaluations had already produced a distinct performance gap on the leaderboard (see Table 4.1), making further trials unlikely to change the final rankings.

Addressing the non-hallucination aspect of RQ4, the evaluation revealed a high-severity technical hallucination regarding the Haskell `concatMap` library function despite the winning configuration's high success rate. Specifically, the system provided the following faulty guidance:

> *"What kind of function does concatMap expect for its second argument: one that takes a single character and returns a list? [...]"*

This hint misidentifies the mapping function's position, contradicting the Haskell

Table 4.1: Blind A/B evaluation results for hints

LLM	Prompt configuration	▲ W	▼ L	≈ T	Win Rate
gpt-5.3-codex	Properties	8	2	6	80%
claude-sonnet-4-6	Baseline	6	5	7	55%
claude-sonnet-4-6	Model solution + properties	7	7	4	50%
gpt-5.3-codex	Baseline	7	7	5	50%
gpt-5.3-codex	Model solution + properties	6	6	4	50%
claude-sonnet-4-6	Model solution	4	4	1	50%
gpt-5.3-codex	Model solution	6	8	4	43%
claude-sonnet-4-6	Properties	2	7	1	22%

Note: Win rate is calculated as $W/(W + L)$, excluding ties from the denominator.

Prelude specification (`concatMap` :: $(a \to [b]) \to [a] \to [b]$) where the mapping function is defined as the first argument [28].

Across all tested LLMs and prompt configurations, other observed issues included:

- **Variable Hallucinations:** Referencing undefined parameters. For example, given the student code:

  ```
  fromBin =
      foldl op 0
      where
          op ? ? =
              ?
  ```

  The system generated a hint referencing a non-existent variable:

  > *"What should happen to the accumulated value 'acc' each time you process a new binary digit? [...]"*

- **Instructional Overreach:** Tells the student the first mechincal step outright. For example, given the student code:

  ```
  simplify x = ?
  ```

  Instead of asking a question or guiding the student, the hint opens with a directive that structures the student's thinking for them:

  > *"First, unpack the input pair so you can work with numerator and denominator separately [...]"*

4.7 User study

The qualitative data gathered from the user study was used in addressing RQ3, with a post-session survey (see Table 3.2) capturing the nuances of the student experience within the AI-integrated Ask-Elle environment.

Findings revealed that while four out of the five participants characterized the terminology of the feedback as confusing, their responses primarily targeted legacy system outputs rather than the GenAI hints. Participants specifically identified property-based counterexamples and type definition errors as sources of confusion, particularly when the tutor provided a "correct" classification for submissions including placeholders, such as myreverse = ?. This lack of clarity in the original Ask-Elle rule-based feedback occasionally hindered the learning flow, as students felt they were not provided with sufficient information to determine their next logical step.

The majority of participants felt that the generative hints successfully respected their original problem-solving intent. Students specifically highlighted that the AI-driven hints were more effective at guiding them toward a solution than the standard feedback mechanisms. Although the system exhibited notable latency during the generation phase, participants generally agreed that it did not break their cognitive flow. While the participants agreed that the system provided sufficient assistance for task completion, many expressed a clear desire for more frequent engagement with the GenAI tutor.

5
Related Work

Traditional research in automated feedback has long emphasized the importance of grounding corrections in verified code. For instance, Singh et al. [29] introduced a system that automatically provides feedback for introductory programming problems by deriving minimal corrections for student solutions. To function effectively, this method requires both a reference implementation of the assignment and an error model consisting of potential corrections for mistakes that students are likely to make. By leveraging these components, the system offers a quantifiable evaluation of a solution's inaccuracies while delivering pedagogical guidance to help students understand their mistakes.

The challenge of unreliable LLM output in educational contexts has been well documented. Azaiz et al. [30] evaluated the performance of GPT-3.5 using 49 unique student submissions selected from two distinct Java programming assignments. To account for the stochastic nature of the model, each submission was processed three times, resulting in a total of 147 feedback instances. The researchers found that GPT-3.5 provided adequate feedback in only 47% of these 147 cases, and mentioned recurring difficulties such as hallucination, failure to recognize correct solutions, and output formatting issues.

Our approach evolves these foundational principles to work with modern generative models. The system employs a hybrid approach that utilizes both Ask-Elle's model solutions and the predictive capabilities of an LLM to propose code completions. These findings directly motivate the gatekeeper architecture in our system, where no hint is presented to a student unless the underlying code has first been verified by the compiler and property-based tests. However, as detailed later in Section 6.5, while the system ensures the functional correctness of the code completion, no formal validation currently exists for the natural language hints themselves.

Closest to our approach is the work of Phung et al. [31], who propose a dual-model architecture in which GPT-4 generates programming hints and a separate GPT-3.5 model validates whether each hint is actionable. While our system shares the same core motivation of ensuring the reliability of feedback before delivery, our gatekeeper validates the underlying code completion rather than the natural language hint itself. This is further discussed as a direction for future work in Section 6.5

The iterative self-correction loop in our pipeline draws on Chen et al. [32],

who demonstrated that LLMs can use compiler errors and failing test output to debug their own generated code across multiple attempts. In our system, this mechanism is the primary means by which the model recovers from initial failures. This ensures that latency from multiple iterations is only ever incurred in service of correctness, never bypassed in favour of speed.

Finally, Roest et al. [33] explored next-step hint generation for introductory programming using LLMs, finding that hints are most effective when constrained to a single actionable observation rather than a list of corrections. This finding directly informed our prompt design, where the model is explicitly prohibited from using additive language and is restricted to producing exactly one guiding question or observation per hint.

6 Discussion

This chapter interprets the findings presented in Chapter 4 and situates them within the broader context of the research questions. While the results demonstrate that the hybrid pipeline achieves consistently high success rates, a deeper analysis reveals meaningful distinctions between the evaluated LLMs that go beyond raw accuracy.

The discussion begins by addressing the discovery of logic defects in the model solutions (Section 6.1). Following this, it examines the trade-offs introduced by LLM scaling, where a smaller architecture's lower inference cost is balanced against an increased reliance on iterative self-correction (Section 6.2). This is followed by a reflection on the final model selection; since the two top-tier LLMs showed nearly identical success rates, the final choice required a qualitative assessment of how well the hints aligned with intended teaching goals (Section 6.3). The chapter concludes by acknowledging the threats to validity that limit these findings (Section 6.4) and identifying directions for future work (Section 6.5).

6.1 Defected Exercises

The discovery of a logic flaw in the `dropevery` model solution highlights a critical vulnerability in tutoring systems, the assumption that model solutions are inherently infallible. To preserve the reliability of the pedagogical process and ensure accurate student assessment, several strategies could be implemented to identify and resolve such defects.

The primary cause of the `dropevery` failure was that model solutions were exempt from the rigorous testing. Within the Ask-Elle framework, if a submission matches a model solution after normalization, it is automatically approved without further verification. To prevent this, model solutions must be subjected to the same QuickCheck property-based testing as AI-generated code. By forcing model solutions through randomized test suites, edge cases, such as the non-positive integers ($n \leq 0$) that triggered infinite recursion, can be identified during the exercise design phase rather than during live deployment.

To fix these logic defects, the exercise description should mention that it excludes problematic edge cases that lack practical meaning. For example, using negative integers in the `dropevery` task is semantically nonsensical. Correspond-

ingly, the property-based test suite should be refined to focus exclusively on valid inputs, preventing the solutions from failing on data it was never intended to process.

6.2 The Iterative Overhead of LLM Scaling

It is interesting that while the "nano" label for GPT-4.1-nano suggests a faster and more efficient experience, the benchmarking data indicates that this did not translate to a quicker remediation time for the student. This duration reasonably depends on the LLM's low first-attempt success rate, which frequently forces the system into an iterative self-correction loop that consumes more time than a single, high-quality generation from a larger model.

From a pedagogical perspective, the significantly higher median NED observed for this LLM is quite revealing. Rather than providing a targeted correction that respects the student's specific logic, the smaller LLM often replaced the student's code with its own interpretation. This behavior suggests that raw generation speed is a poor substitute for the reasoning depth required to properly align with a student's problem-solving intent. Furthermore, the observation that configurations including a model solution perform better demonstrates that the AI, in many cases, directly incorporates the model solutions into its output. This reliance on the model solution can potentially result in feedback that, while technically valid, fails to align with the student's implementation strategy.

6.3 Final Model Selection

In evaluating the performance of the LLMs, Claude Sonnet 4.6 and GPT-5.3-Codex, it is interesting to note that their quantitative results were nearly indistinguishable across several primary metrics. As shown in the benchmarking data, both LLMs achieved high terminal success rates. Furthermore, their median remediation latencies were also similar, generally falling within a narrow window of 4.1 to 5.0 seconds. Because the rate of generating technically valid solutions and the time required to do so were comparable, these metrics could not serve as the primary basis for selecting a final LLM.

Instead, the selection process required a deeper analysis of how each LLM engaged with the student's code, specifically evaluating code preservation and the clarity and depth of the pedagogical hints. A key differentiator emerged in the NED data. It is interesting that GPT-5.3 Codex consistently demonstrated a more conservative approach to code completion, achieving a median NED of 0.39 in the Properties configuration. In contrast, Claude Sonnet 4.6 showed a higher tendency to restructure the student's code on all of its configurations. However, NED can be a misleading measure in isolation. Because it calculates edit distance at the character level, trivial stylistic changes, such as renaming a long variable to a shorter one, can result in a higher NED score than a fundamental algorithmic

shift. A prime example is the distinction between `foldl` and `foldr`. Textually, the difference is a single character, however semantically, they represent entirely different computational strategies:

`foldl` is a higher-order function that reduces a list to a single value by processing elements from left to right.

$$\texttt{foldl}\ (-)\ 0\ [1,2,3] \implies ((0-1)-2)-3 = -6$$

`foldr` is a higher-order function that reduces a list to a single value by processing elements from right to left.

$$\texttt{foldr}\ (-)\ 0\ [1,2,3] \implies 1-(2-(3-0)) = 2$$

Changing a student's `foldl` to `foldr` would result in a negligible NED impact, yet it fundamentally overrides the student's chosen recursion pattern and alters the output. Because NED operates purely at the character level, insensitive to such profound semantic shifts, this limitation makes it difficult to determine, based solely on quantitative metrics, which LLM most accurately preserves the student's strategy. Furthermore, due to the limited timeframe of this research, it was not possible to conduct a comprehensive manual audit to determine the frequency of these semantic overrides.

Reasonably, the pedagogical value of a hint depends on its ability to meet a student where they are in their own cognitive process. During the Blind A/B evaluation, we observed that the hints generated by the Codex Properties configuration were more instructionally effective. This was reflected in the leaderboard results, where this specific configuration recorded 8 wins and only 2 losses. From the perspective of the researchers (who are students themselves), the generations provided by Codex felt more supportive and less intrusive. Due to the clarity of its guiding hints, we subjectively concluded that Codex offered a better user experience. Ultimately, the decision to select GPT-5.3 Codex in its properties configuration as the final model was based on this balance of technical reliability and a perceived higher quality of instructional guidance.

6.4 Threats to Validity

A significant threat to the construct validity of this research concerns the measurement of system latency. As established in Section 4.1, the logging mechanism implemented in the current prototype captures the duration of the GenAI inference phase exclusively. However, this metric does not represent the end-to-end throughput, the total elapsed time from the moment a student submits their Haskell code until a pedagogical hint is rendered in the user interface. Consequently, the reported execution times represent a lower-bound estimate of the GenAI's processing rather than a comprehensive measure of the system's real-time pedagogical efficacy.

The group of people who took part in the study also creates some concerns

regarding the results. The main issue is the small number of participants. Due to the limited sample size, it is difficult to generalize these findings to the wider population of students. Another problem is that the participants had very different levels of experience with Haskell. A student with more experience might be able to use a hint generated by GenAI to solve a task, while a beginner might still find the same hint confusing.

A further threat to conclusion validity concerns the benchmarking. Although 36 student submissions were evaluated, many exercises were solved with 100% consistency, meaning the effective sample size for testing the system's performance is smaller than it seems. Because LLMs are probabilistic, their performance and latencies can fluctuate even when given the same prompt. Running each test multiple times would have accounted for this stochastic variance and provided a more robust statistical mean for success rates.

Finally, a threat to conclusion validity arises from the subjective nature of the model selection process. Although a Blind A/B evaluation was conducted to mitigate bias, we acted as the evaluators. Our own expertise in Haskell likely influenced which hints were deemed "high-value instructional hint". A beginner student might have different preferences or needs that were not fully captured by the our evaluation. Additionally, the use of NED as a proxy for pedagogical quality assumes that minimal intervention is always superior. However, there may be cases where a more extensive rewrite is necessary to correct a fundamental misconception.

6.5 Future Work

The integration of GenAI into Ask-Elle has demonstrated significant potential to expand the coverage of automated feedback, yet the current implementation serves as a foundation rather than a final solution. Future efforts should prioritize the refinement of the validation architecture, solutions for optimization and stress-testing the system against more rigorous pedagogical challenges.

Automated Verification of Pedagogical Content
A primary concern identified during the evaluation was the hallucination between the verified code and the natural language hint. While the system effectively ensures that the generated Haskell code satisfies all formal properties, it currently lacks a mechanism to verify that the textual advice accurately describes that code. The occurrence of hint inaccuracies, such as the misstatement of the `concatMap` type signature, suggests that future work should implement a secondary verification layer. This could involve implementing an automated validation tool to ensure that any technical terms or type signatures mentioned in the text are cross-referenced with the Haskell `Prelude`. An alternative validation strategy would be to feed the generated natural language hint back into the LLM as a prompt, instructing it to produce a code completion based solely on the hint's guidance. The resulting code could then be compared against the already verified completion to assess whether the hint

describes the correct solution.

Latency Optimization and Adaptive Model Switching
While participants in the user study indicated that the system's latency did not break their cognitive flow, it was nonetheless notable, and the response times for the submissions remain an operational challenge. Future research should investigate methods to reduce this overhead without sacrificing the reliability provided by the validation loop. One potential solution is the implementation of adaptive model switching, where a smaller, more efficient LLM like GPT-4.1-nano is used for simple hole-filling tasks, while a more robust LLM like GPT-5.3 Codex is reserved for high-complexity scenarios. Additionally, implementing a feedback cache, where verified solutions for common Discarded patterns are stored and reused, could significantly improve responsiveness for the most frequent student errors.

Evaluating Performance on High-Complexity Tasks
While the LLMs currently handle introductory exercises with high precision, it remains to be seen where their limit lies when faced with advanced exercises. This can be achieved by manually designing advanced exercises with corresponding model solutions and properties, then replicating the methodology used in this research to identify where performance begins to degrade.

7
Conclusion

The primary objective of this thesis was to investigate whether integrating GenAI into the Ask-Elle tutoring system could enhance the quality and coverage of automated feedback for students learning functional programming. The project successfully designed, implemented, and evaluated a hybrid architecture.
The findings derived from the development and evaluation of this prototype provide definitive answers to the research questions established at the outset of this work:

- Integration and Validation (RQ1 & RQ2): This research demonstrates that GenAI can be effectively integrated into Ask-Elle's existing strategy-based and property-based mechanisms by employing a validation loop architecture. By positioning Ask-Elle as a gatekeeper, the system ensures that any pedagogical hint presented to the student is grounded in a functionally correct, machine-verified code completion. The iterative self-correction loop, where compiler and testing logs are fed back to the model, was proven to be a robust mechanism for ensuring technical reliability.

- Pedagogical Improvement (RQ3): The combined system significantly improves the coverage of automated feedback by successfully addressing submissions classified as Discarded, which previously received no meaningful guidance. Data from the user study suggests that these AI-enhanced hints provide higher educational value than legacy feedback.

- Reliability (RQ4): The AI-enhanced prototype is remarkably reliable in terms of code generation, achieving success rates exceeding 94% across the most effective model and prompt configurations. While the automated gatekeeper successfully prevents the delivery of incorrect code, rare instances of technical hallucinations in natural language hints indicate that the textual layer remains the only remaining vulnerability in the trust chain.

In summary, this work establishes that a hybrid approach, combining the probabilistic reasoning of LLMs with the deterministic rule-based system, results in a more adaptive ITS. This integration effectively transforms previously Undecided student submissions into constructive learning opportunities, marking a step forward in automated functional programming education.

Bibliography

[1] H. Keuning, J. Jeuring, and B. Heeren, "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, Sep. 2018. [Online] Available: https://doi.org/10.1145/3231711.

[2] F. Martin, M. Zhuang, and D. Schaefer, "Systematic Review of Research on Artificial Intelligence in K-12 Education (2017–2022)," *Computers and Education: Artificial Intelligence*, vol. 6, p. 100195, Jun. 2024, doi: https://doi.org/10.1016/j.caeai.2023.100195.

[3] M. Su, L. Ma, D. Zhang, A. Yunusa-Kaltungo and C. Cheung, "Exploring Benefits and Concerns of Incorporating Digital Tools into Engineering Education," *European Journal of Education and Pedagogy*, vol. 6, no. 1, pp. 45–51, Jan. 2025, doi: https://doi.org/10.24018/ejedu.2025.6.1.909.

[4] A. Gerdes, B. Heeren, J. Jeuring and L. T. van Binsbergen, "Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback," *International Journal of Artificial Intelligence in Education,* vol. 27, no. 1, pp. 65-100, Mar. 2017, doi: https://doi.org/10.1007/s40593-015-0080-x.

[5] Y. Zhang et al, "Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models," *Computational Linguistics*, vol. 51, no. 3, pp. 1373–1418, Dec. 2025, doi: https://doi.org/10.1162/COLI.a.16.

[6] Helium, Version 1.8.1, [Software], Utrecht, Netherlands : Utrecht University, 2015. [Online]. Available: https://hackage.haskell.org/package/helium.

[7] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning haskell," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Uppsala, Sweden, 2003, pp. 62–71. [Online] Available: https://doi.org/10.1145/871895.871902.

[8] GHC, Version 9.14.1, [Software], Glasgow, UK : The GHC Team, 2025. [Online]. Available: https://www.haskell.org/ghc/.

[9] S. Marlow and S. Peyton Jones, "The Glasgow Haskell Compiler," in *The Architecture of Open Source Applications*, vol. 2, A. Brown and G. Wilson, Eds. Lulu, 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler.

[10] S. B. Blessing, "A Programming by Demonstration Authoring Tool for Model-Tracing Tutors," in *Authoring Tools for Advanced Technology Learning Environments: Toward Cost-Effective Adaptive, Interactive and Intelligent Educational Software*, T. Murray, S. B. Blessing, and S. Ainsworth, Eds. Dordrecht, Netherlands: Kluwer Academic Publishers, 2003, ch. 4, pp. 93–119. [Online]. Available: https://doi.org/10.1007/978-94-017-0819-7_4, Accessed on: 12 May

2026.
[11] J. R. Anderson and E. Skwarecki, "The automated tutoring of introductory computer programming," *Commun. ACM*, vol. 29, no. 9, pp. 842–849, Sep. 1986. [Online]. Available: https://doi.org/10.1145/6592.6593.
[12] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00),* Montreal, Canada, 2000, pp. 268-279. doi: https://doi.org/10.1145/357766.351266.
[13] R. Krook, N. Smallbone, B. J. Svensson, and K. Claessen, "Quickercheck: Implementing and evaluating a parallel run-time for quickcheck," in *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*, ser. IFL '23. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3652561.3652570
[14] L. Banh and G. Strobel, "Generative artificial intelligence," *Electron. Markets*, vol. 33, no. 1, p. 63, Dec. 2023. [Online]. Available: https://doi.org/10.1007/s12525-023-00680-1.
[15] Artificial Analysis, "Artificial Analysis Intelligence Benchmarking Methodology," 2026. [Online]. Available: https://artificialanalysis.ai/methodology/intelligence-benchmarking (accessed on: 2026-05-20).
[16] GPT-5.3 Codex, Version 5.3, [Software], San Francisco, USA : OpenAI, 2026. [Online]. Available: https://openai.com/sv-SE/index/introducing-gpt-5-3-codex/.
[17] "Pricing," OpenAI, 2026. [Online]. Available: https://developers.openai.com/api/docs/pricing. [Accessed: May 6, 2026].
[18] Artificial Analysis, "GPT-5.3 Codex (xhigh) Intelligence, Performance Price Analysis," 2026. [Online]. Available: https://artificialanalysis.ai/models/gpt-5-3-codex (accessed on: 2026-05-19).
[19] GPT-4.1, Version 4.1, [Software], San Francisco, USA : OpenAI, 2025. [Online]. Available: https://openai.com/index/gpt-4-1/.
[20] Artificial Analysis, "GPT-4.1 nano Intelligence, Performance Price Analysis," 2026. [Online]. Available: https://artificialanalysis.ai/models/gpt-4-1-nano (accessed on: 2026-05-19).
[21] Anthropic, "Introducing Claude Sonnet 4.6," Feb. 17, 2026. [Online]. Available: https://www.anthropic.com/news/claude-sonnet-4-6. [Accessed: May 6, 2026].
[22] Anthropic, "Claude Pricing," 2026. [Online]. Available: https://platform.claude.com/docs/en/about-claude/pricing (accessed on: May 12, 2026).
[23] Artificial Analysis, "Claude Sonnet 4.6 (Adaptive Reasoning, Max Effort) Intelligence, Performance Price Analysis," 2026. [Online]. Available: https://artificialanalysis.ai/models/claude-sonnet-4-6-adaptive (accessed on: 2026-05-19).
[24] GitHub, "Introduction to dev containers," GitHub Documentation. [Online]. Available: https://docs.github.com/en/codespaces/setting-up-your-project-for-codespaces/adding-a-dev-container-configuration/introduction-to-dev-containers [Accessed: May 20, 2026].
[25] Ollama, "Models," *Ollama*, 2026. [Online]. Available: https://ollama.com/search.

[26] K. R. Koedinger and V. Aleven, "Exploring the Assistance Dilemma in Experiments with Cognitive Tutors," Educ. Psychol. Rev., vol. 19, no. 3, pp. 239-264, Sep. 2007, doi: 10.1007/s10648-007-9049-0.

[27] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[28] S. Marlow, Ed., "Haskell 2010 Language Report," 2010. [Online]. Available: https://www.haskell.org/definition/haskell2010.pdf.

[29] R. Singh, S. Gulwani and A. Solar-Lezama, "Automated Feedback Generation for Introductory Programming Assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13),* Seattle, WA, USA, 2013, pp. 15-26. doi: https://doi.org/10.1145/2499370.2462195.

[30] I. Azaiz, O. Deckarm, and S. Strickroth, "AI-Enhanced Auto-Correction of Programming Exercises: How Effective is GPT-3.5?," *Int. J. Eng. Pedagog.*, vol. 13, no. 8, pp. 67–83, Dec. 2023. [Online]. Available: https://doi.org/10.3991/ijep.v13i8.45621.

[31] T. Phung, V.-A. Pădurean, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, "Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation," in *Proceedings of the 14th Learning Analytics and Knowledge Conference (LAK '24)*, Kyoto, Japan, 2024, pp. 1–12. doi: https://doi.org/10.1145/3636555.3636846.

[32] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," 2023. [Online]. Available: https://arxiv.org/abs/2304.05128

[33] L. Roest, H. Keuning, and J. Jeuring, "Next-Step Hint Generation for Introductory Programming Using Large Language Models," in *Proceedings of the 26th Australasian Computing Education Conference (ACE '24)*, Sydney, Australia, 2024, pp. 1-11. [Online]. Available: https://doi.org/10.1145/3636243.3636259, Accessed on: May 12, 2026.

A

Appendix 1

Listing A.1: Initial code generation prompt.

```haskell
callAIForCode :: PromptConfig -> ExerciseContext -> Exercise a -> a -> IO String
callAIForCode cfg ec ex term = do
  let prompt = unlines $
        [ "You are an automated Haskell code-completion engine. Your only function is to output valid Haskell code based on instructions."
        , "The student submission contains 'holes' represented by the '?' character."
        , ""
        , "### INSTRUCTION HIERARCHY (Follow strictly):"
        , "1. PRIORITY 1: Fill the '?' holes with valid Haskell expressions to complete the solution."
        , "2. Keep all existing student code EXACTLY as it is. Only replace the '?' characters."
        , "3. ONLY if a correct solution is logically impossible by just filling the holes, you may perform a minimal repair of the surrounding code."
        , "4. LAST RESORT: Change the algorithmic strategy (e.g., switching from recursion to folds) ONLY if the student's current approach is fundamentally incapable of solving the exercise."
        , ""
        , "### CONTEXT"
        , "Exercise description: " ++ ecDescription ec
        , ""
        ]
        ++ (if useQuickCheck cfg then ["QuickCheck properties:\n", ecProperties ec, ""] else [])
        ++ (if useModels cfg then ["Model Solutions (for reference):\n", ecModelSols ec, ""] else [])
        ++ [ "### STUDENT SUBMISSION"
           , prettyPrinter ex term
           , ""
           , "Output the completed Haskell code enclosed strictly within <haskell_code> and </haskell_code> tags."
           , "Do not output any text, explanations, or greetings outside of these tags."
           , ""
```

```
25            , "<haskell_code>"
26            ]
27  logAI "GENERATE CODE PROMPT" prompt
28  callLLM prompt
```

Listing A.2: Prompt to generate new code based on errors that occured in the previous code.

```
callAIWithFeedback :: PromptConfig -> ExerciseContext -> String ->
    Exercise a -> a -> String -> IO String
callAIWithFeedback cfg ec feedback ex term code = do
    let prompt = unlines $
            [ "You are an automated Haskell code-repair engine. Your only
    function is to output valid Haskell code based on instructions."
            , "Your previous attempt to fix the student's code failed."
            , ""
            , "### FAILURE REPORT"
            , "The previous code generated a test failure or error: " ++
    feedback
            , "You MUST produce code that differs meaningfully from your
    previous attempt."
            , ""
            , "### INSTRUCTION HIERARCHY (Correct the mistake following
    these rules):"
            , "1. PRIORITY 1: Fill the '?' holes in the student's
    original structure."
            , "2. FIX THE ERROR: Address the failure reported above
    without breaking the overall structure."
            , "3. MINIMAL REPAIR: Only change student code if the current
    structure is fundamentally broken."
            , ""
            , "### CONTEXT"
            , "Exercise: " ++ show (getId ex)
            , "Description: " ++ ecDescription ec
            ]
            ++ (if useQuickCheck cfg then ["QuickCheck properties:\n" ++
    ecProperties ec] else [])
            ++ (if useModels cfg then ["Model Solutions (for
    reference):\n" ++ ecModelSols ec] else [])
            ++ [ ""
            , "### PREVIOUS ATTEMPT (FAILED: do NOT copy or reproduce
    this)"
            , "The following code was rejected. Your output must NOT look
    like this:"
            , code
            , ""
            , "### STUDENT SUBMISSION (Your ONLY valid starting point)"
            , "Start from THIS code. Do not use the failed attempt above
    as a base:"
            , prettyPrinter ex term
            , ""
            , "Output the corrected Haskell code enclosed strictly within
    <haskell_code> and </haskell_code> tags."
            , "Do not output any text, explanations, or greetings outside
    of these tags."
```

```
33              , ""
34              , "<haskell_code>"
35              ]
36      logAI "FEEDBACK/RETRY PROMPT" prompt
37      callLLM prompt
```

Listing A.3: Prompt for generating hint.

```
callAIForHint :: Exercise a -> String -> a -> IO String
callAIForHint ex aiCode term = do
    let prompt = unlines
            [ "### ROLE"
            , "You are a supportive Haskell tutor. Your only job is to give the student ONE small nudge toward their next immediate step."
            , ""
            , "### CONTEXT"
            , "Student's current (incomplete or incorrect) code:"
            , prettyPrinter ex term
            , ""
            , "Reference correct solution (for your eyes only - never reveal it):"
            , aiCode
            , ""
            , "### TASK"
            , "Identify the single most important thing the student should fix or think about RIGHT NOW."
            , "Give exactly one hint about that one thing. Stop there."
            , "Do not mention any other issues, even if you can see them."
            , ""
            , "### STRICT CONSTRAINTS"
            , "- ONE hint only. If you find yourself writing 'also' or 'additionally', stop and delete everything after the first point."
            , "- Do NOT give a hint about step 2 if they have not solved step 1."
            , "- Do NOT provide code snippets or reveal the solution."
            , "- Do NOT use Markdown."
            , "- Maximum 2-3 sentences."
            , "- Phrase the hint as a single guiding question or a single observation."
            ]
    logAI "HINT PROMPT" prompt
    callLLM prompt
```

DEPARTMENT OF COMPUTER SCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY