communicated by:
Lehr- und Forschungsgebiet Informatik 9

Prof Dr. Ulrik Schroeder



The present work was submitted to the Learning Technologies Research Group
Diese Arbeit wurde vorgelegt am Lehr- und Forschungsgebiet Lerntechnologien

Extension and Evaluation of Feedback Mechanisms in the Python Repair Tool Pyrat through the Use of LLMs

Erweiterung und Evaluation von Feedbackmechanismen im Python-Reparatur-Tool Pyrat durch den Einsatz von LLMs

Bachelor-Thesis
Bachelorarbeit

presented by / von

Herr, Yuval
433722

Examined by / Begutachtet von

Prof. Dr. Ulrik Schroeder
Prof. Dr. Jürgen Giesl

Supervised by / Betreut von

Annabell Brocker

Aachen, September 4, 2025

Contents

List of Figures iii

List of Listings iv

1 Introduction 2

2 Related Work 4
2.1 Code Assessment Tools . . . 4
2.2 Code Repair Tools . . . 5
2.3 Feedback Generation with LLMs . . . 6
2.4 Session-Aware LLM Feedback . . . 7

3 Conceptual Approach and Implementation 8
3.1 Session Context . . . 8
3.1.1 Rationale and Objectives . . . 8
3.1.2 Technical Implementation . . . 10
3.2 Extensions to Feedback Generation . . . 14
3.2.1 Conceptual Integration into the Existing Feedback Generation . . . 14
3.2.2 Feedback Methods and Refinements . . . 15
3.2.3 Rubber Duck Debugging as a Feedback Mechanism . . . 16
3.2.4 Code-Level and Architectural Integration . . . 18
3.2.5 Prompt Construction for Session-Based Feedback Generation . . . 19

4 Evaluation 27
4.1 Building the Questionnaire . . . 27
4.1.1 Selecting the Tasks . . . 28
4.1.2 Evaluation Questions . . . 29
4.2 Evaluating the Results . . . 31
4.2.1 Quantitative Results . . . 31
4.2.2 Qualitative Results . . . 33
4.2.3 Limitations . . . 35
4.2.4 Interpretation of Results . . . 37

5 Future Work 39

6 Conclusion 41

Appendix 42

A Bibliography **43**

B Evaluation Codes **46**

C Swagger Documentation of `/api/check/session` Endpoint **50**

D Further Evaluation Results **52**

E Digital Appendix **55**

List of Figures

3.1 Schema representation of the session structure 11
3.2 High-Level Activity Diagram of Session Handling Prior to Feedback Generation 12
3.3 High-level overview of the adapted Pyrat workflow based on the high-level overview diagram in from the master's thesis by Steffes [Ste24] . . 14

4.1 Evaluation Questions 29
4.2 Example question for the first part of the evaluation 30
4.3 Survey Questions 31
4.4 Evaluation Results for Task Vehicle Class 32
4.5 Evaluation Results for Task ToDoList 34

List of Listings

3.1 General context prompt used in `llmPrompts.ts` 20
3.2 Safety instructions prompt used in `llmPrompts.ts` 20
3.3 Standard prompt for LLM-based feedback without refinements 22
3.4 Prompt used for the `select` feedback refinement 22
3.5 Prompt used for the `rubber duck` feedback refinement 23
3.6 Full response format prompt including strict output rules 24

Abstract

As the demand for programming education grows and access to individualized instruction remains limited, automated feedback systems are becoming increasingly important. Pyrat is a web-based application originally developed by Steffes to support students in validating and repairing Python code through structured feedback. It emphasizes student autonomy and uses an Abstract Syntax Tree (AST)-based analysis to generate rule-based corrections. While this enables interpretable and controlled feedback, it often produces low-level messages and struggles to scale to more complex tasks, resulting in excessive correction steps.

This thesis extends Pyrat with several improvements to enhance feedback clarity and usefulness. First, a session context mechanism was introduced to persist user submissions and received feedback over time, aiming to enable more coherent and individualized interactions. Second, feedback generation was augmented with Large Language Models (LLMs), prompted using intermediate results from Pyrat—such as the closest reference solution or buggy code regions—to combine rule-based precision with LLM flexibility. These methods were further refined through prompt engineering and a rubber duck debugging approach, where the LLM encourages self-explanation by asking targeted questions based on identified issues.

The evaluation compared feedback strategies in terms of perceived usefulness and clarity. All LLM-based methods outperformed the original rule-based approach, producing more consistent and higher-rated feedback. While some inaccuracies occurred, such as hallucinated details, their impact was minimal. Rubber duck debugging was less well received, potentially due to limited evaluation scope, mismatched expectations, or model deviation under complex prompts.

The findings show that LLMs are a valuable addition to student-centered feedback systems and perform well even in resource-efficient settings. They also highlight the importance of precise prompting, suitable model selection, and effective context management. Future work should focus on improving robustness, better understanding the role of reflective feedback in appropriate learning contexts, and leveraging stronger reasoning models for more complex tasks and proactive strategies.

Chapter 1 Introduction

Automatically generated feedback is a key component in programming education, especially for novices who lack experience in interpreting compiler errors or debugging their own code. While several educational tools provide basic validation or example-based hints, few systems offer corrective feedback that is both precise and pedagogically effective. This thesis builds upon the tool *Pyrat*, originally developed by Leonard Steffes [Ste24], which enables learners to work through programming tasks independently and receive automated guidance based on structural code comparisons.

Pyrat was designed to combine validation and repair into a unified system, offering students textual correction steps that transform incorrect submissions into valid solutions. If a submission is already correct, it is detected during validation and no repair is applied. Although Pyrat produces technically accurate feedback, it has limitations in terms of clarity and perceived usefulness. Evaluations of the original version revealed that the feedback was often overwhelming or hard to understand for beginners—particularly in complex submissions. This was partly due to the low level of abstraction in the correction steps, which stems from the underlying AST-based repair mechanism. The issue was further amplified when the learner's code was structurally distant from the selected reference solution, leading to a large number of isolated changes. Moreover, the system generated feedback statelessly, without taking into account previous learner interactions or progress over time [Ste24].

Importantly, the evaluation in the original thesis also showed that users were generally willing to accept longer feedback generation times when the quality of the feedback improved. This finding supports the feasibility of integrating more sophisticated methods—such as LLM-based feedback generation—despite the potential for increased response times [Ste24].

This thesis addresses the aforementioned limitations by rethinking Pyrat's feedback generation process. First, it repurposes existing intermediate outputs from the repair algorithm—this includes the closest determined reference solution, identified faulty nodes in the AST, and correction steps—as structured input for the language model. The LLM used in this thesis is *gpt-4o-mini* by OpenAI[1]. It generates feedback in natural language, guided by carefully constructed prompts that remain aligned with the underlying repair logic. Second, the system now retains a simple session context across interactions, allowing feedback to adapt based on recent submissions. By persisting prior submissions and the received feedback, the system can adjust its

[1] `https://openai.com`, last accessed 31/03/2025

responses based on a learner's recent history. Additional refinements allow for variation in feedback style, such as prompting reflective thinking through Rubber Duck Debugging.

While these changes introduce a new level of flexibility and expressiveness to Pyrat, the system remains modular: it can operate without LLMs, falling back to deterministic feedback generation. This ensures reproducibility and addresses concerns expressed by Steffes regarding dependency on proprietary models [Ste24]. The central concern of this thesis is therefore not just technical integration, but the exploration of how different feedback methods affect perceived learning support.

The central research question of this thesis is therefore: *How do different feedback generation methods in Pyrat compare in terms of perceived usefulness and clarity?* To explore this, the introduced methods are evaluated not only technically, but also through the lens of user experience. Can LLM-based feedback provide clearer or more helpful guidance than the original rule-based approach? Does reflective prompting enhance understanding, or does it risk overwhelming learners? These guiding questions shape both the design and evaluation of the system.

This work is structured as follows: Chapter 2 reviews relevant systems and research in automated feedback and educational tooling. Chapter 3 presents the conceptual and technical extensions introduced in this thesis, including session-based context handling and prompt-driven feedback refinement. Chapter 4 addresses the central research question based on a combined quantitative and qualitative evaluation. Chapter 5 identifies remaining limitations and outlining possible directions for future development and Chapter 6 summarizes the findings and concludes the thesis.

Chapter 2 Related Work

Recent advances in programming education have led to a range of tools and systems for automatically evaluating student code, providing repair suggestions, and generating feedback. These developments span from rule-based assessment and repair mechanisms to newer approaches that incorporate large language models and interaction context.

This chapter reviews prior work across four areas relevant to this thesis: code assessment tools, automated code repair systems, LLM-based feedback generation, and session-aware feedback mechanisms. Each section introduces representative systems and studies, discusses their capabilities and limitations, and situates the contributions of this thesis within the broader research landscape. Special attention is given to the Pyrat system, which forms the basis for this work and is extended here to generate more adaptive and context-aware feedback.

2.1 Code Assessment Tools

Code assessment tools are widely used in programming education to evaluate student submissions automatically. They typically rely on static and dynamic analysis to check for correctness, often through predefined unit tests. The feedback provided by such systems usually consists of binary results (e.g., “correct” or “incorrect”) and, depending on the implementation, basic test outputs. While this approach is scalable and effective for identifying functional correctness, it often falls short in supporting deeper learning processes, particularly for novices [KN16].

A commonly used tool in this category is *nbgrader*, a plugin for JupyterLab that enables the creation, distribution, and automatic grading of programming assignments [BBB+19] [Jup24]. Instructors define exercises using Markdown and Python code cells, including hidden grading cells that run test cases on student submissions. The system follows an instructor-centered assessment model, where feedback is typically only accessible after grading has been completed. As a result, students often receive feedback in the form of test pass/fail indicators without further explanation or instructional support. This can lead to difficulties in understanding mistakes, especially for beginners who are unfamiliar with debugging strategies.

The limitations of such systems have been discussed in the literature. Kyrilov and Noelle [KN16] observed that minimal feedback—limited to correctness alone—led to low correction rates among students after an initial failed attempt. In their study, only 43% of students who received an "incorrect" result were able to resolve their issue within five minutes. The authors argue that this form of feedback is insufficient for effective learning, particularly in early programming education.

Some recent approaches have experimented with extending assessment tools using large language models to evaluate student explanations or to classify open-ended responses. For instance, Oli et al. [OBCR23] explored the use of GPT-based models to automatically assess student-written explanations of code, finding that such models can approximate expert grading in specific contexts. However, these approaches still focus on assessment and correctness classification, rather than on generating formative, instructional feedback.

In summary, while code assessment tools provide efficient mechanisms for correctness evaluation, they typically lack pedagogical depth. Although emerging methods involving LLMs offer new possibilities for more nuanced assessment, the feedback remains primarily evaluative in nature, not explanatory or adaptive. This underlines the need for complementary approaches that generate instructional feedback beyond correctness alone.

2.2 Code Repair Tools

In contrast to code assessment tools, code repair tools do not return whether a student's code is correct or incorrect but provide feedback on how the student's code can be repaired, i.e., transformed into a corrected version that meets the instructor's predefined criteria for correctness, such as unit tests, in cases where errors occur.

Pyrat is a tool for automated validation and repair of Python code submissions, originally developed to support novice programmers by identifying incorrect code and suggesting minimal changes to produce a correct version [Ste24]. The system performs both static and dynamic validation: code is first checked for syntax correctness and basic structural constraints, followed by semantic validation through unit tests. If a submission passes these formal checks but fails semantic validation, a repair process is initiated.

At the core of Pyrat's repair mechanism is an algorithm based on the *All Path Tree Edit Distance* (APTED) [PA16], which compares the abstract syntax tree (AST) of the student's submission to one or more reference solutions. The algorithm identifies a minimal sequence of edits needed to transform the incorrect AST into a correct one. These edit operations are then used to derive suggestions for repairing the student's code. Since Pyrat relies on AST comparison, the submitted code must be syntactically correct before any repair can be performed [Ste24].

In contrast to earlier systems such as CLARA [GRZ18] and Refactory [HAM+19], Pyrat does not depend on a large corpus of correct student submissions or extensive code refactoring to generate repair suggestions. Instead, instructors may provide one or more correct reference implementations manually or generate them using an LLM. This reduces the overhead associated with preparing exercises and improves the system's flexibility. Furthermore, Pyrat supports a broader range of Python features than earlier tools, including object-oriented constructs and common libraries, which increases its applicability in realistic programming courses.

While Pyrat effectively integrates validation and repair, its outputs are still focused on structural corrections and not inherently designed to function as instructional feedback. The suggestions it produces describe what to change in the code, but not necessarily how or why—a distinction that becomes central in the context of generating explanatory feedback for learners.

2.3 Feedback Generation with LLMs

While conventional automated repair systems focus on transforming incorrect code into a correct version, recent research has explored the use of large language models to generate natural-language feedback that helps learners understand and resolve their mistakes. In this context, feedback is not limited to code modifications but includes explanations, hints, or reflections intended to support the learning process. This distinction is particularly relevant for novice programmers, who may benefit more from understanding error causes than from simply receiving corrected code.

Kazemitabaar et al. introduced *CodeAid*, an LLM-powered educational assistant designed to support programming students without directly providing solutions [KYW+24]. The system is capable of generating pseudocode explanations, commenting on problematic code segments, and offering conceptual guidance. In a deployment involving over 700 students, CodeAid was used thousands of times and was perceived by instructors and students as providing helpful and appropriately calibrated feedback. Notably, the assistant was designed to encourage cognitive engagement by withholding full solutions and instead offering stepwise support.

The importance of structuring LLM prompts to produce pedagogically meaningful feedback has also been emphasized in recent work. Lohr et al. investigated whether LLMs can generate specific types of feedback aligned with educational taxonomies, such as hints, explanations, or reflections [LKK25]. Their findings suggest that LLMs can be prompted to produce targeted feedback categories with varying success, depending on prompt structure and task complexity. This supports the idea that feedback quality can be influenced by prompt engineering and opens possibilities for adapting feedback to different learning stages.

In addition to content type, the timing and immediacy of feedback are important factors. Woodrow et al. evaluated a style-focused feedback system powered by GPT-4 that was integrated into an online introductory course with thousands of students [WMP24]. The system delivered comments on code readability and clarity shortly after submission. The study reported increased engagement with feedback and measurable improvements in code quality among students who interacted with the system. These results indicate that LLMs can be used not only for correctness-related feedback but also for improving style and fostering iteration.

Further approaches have explored multi-level or scaffolded feedback. For example, Xiao et al. proposed a method of generating feedback at varying levels of specificity, ranging from abstract hints to concrete code suggestions [XHS24]. Their evaluation showed that students benefit from starting with general guidance, but often require more specific feedback as they continue to struggle. Such tiered strategies align with pedagogical models that emphasize gradually increasing support and are particularly relevant when using LLMs in interactive environments.

Collectively, these studies suggest that LLMs can be used to generate a range of feedback types that go beyond simple correctness checks or structural repairs. However, most existing systems operate on a per-submission basis and do not retain contextual information across multiple attempts. The integration of session history and personalized context into LLM-generated feedback remains limited—an aspect addressed in the following section.

2.4 Session-Aware LLM Feedback

While many systems generate feedback based solely on the current code submission, recent research has highlighted the potential of incorporating interaction context to improve the relevance and pedagogical quality of automated feedback. In this setting, session awareness refers to the ability of a system to consider prior interactions—such as previous submissions, feedback messages, or user behavior—when generating responses.

Heickal et al. propose the concept of a *feedback ladder*, in which multiple tiers of feedback are generated for a given task [HL24]. Instead of presenting a complete explanation immediately, the system progressively reveals more detailed feedback based on learner need. While this approach does not persist full session history, it reflects a structured use of sequential context to guide learning. Similarly, Xiao et al. explore multi-level LLM-generated feedback that ranges from general hints to code-level suggestions, with studies showing that such scaffolding improves learning outcomes for novice programmers [XHS24].

Some approaches also explore broader personalization through user modeling. For example, Tang et al. introduce a framework that allows instructors to modify and contextualize LLM-generated feedback across many students [TWH+24]. While this system does not track fine-grained session history, it supports feedback variation based on student profiles, suggesting that adaptation to learner characteristics is both feasible and pedagogically useful.

Although these systems differ in technical design and use case, they share the goal of moving beyond isolated feedback instances toward context-informed interactions. The incorporation of session context into feedback generation is an emerging trend that shows promise for improving engagement, clarity, and progression in learning environments. This thesis contributes to this direction by applying session-based context retention in the domain of Python repair tools, building upon but not replicating the methods explored in prior work.

Chapter 3 Conceptual Approach and Implementation

This chapter introduces the conceptual and technical foundations of the extensions made to Pyrat as part of this thesis. The first section presents the implementation of session context, which serves as the foundation for the extended feedback generation methods. The second section discusses these feedback extensions in detail, covering the various strategies developed, their refinements, and how they interact with session context.

Each component is introduced with a brief conceptual motivation, framed by the overarching research question, followed by a detailed explanation of the corresponding implementation. This structure ensures that the rationale behind each change is clearly presented alongside its technical realization.

3.1 Session Context

The first part of this thesis involved implementing session context functionality for Pyrat. This addition allows a user's submission history to be persisted in the backend, primarily storing the code submitted by the user along with the corresponding feedback received. This functionality was integrated into Pyrat's existing feedback system and serves as the foundation for context-based feedback generation using large language models.

To support this feature, a new endpoint was added to Pyrat's API, complementing the existing `/api/check` endpoint. This ensures that the original, stateless interaction mode—where no user-specific information is persisted—remains available and functional.

3.1.1 Rationale and Objectives

Session management is a fundamental component in transitioning from a static, rule-based feedback system to a more adaptive and context-aware approach. In the previous implementation, feedback was generated deterministically via a REST API, where each request was processed in isolation and identical responses were produced for the same input. While this ensured consistency, it limited the system's ability to adapt to prior user interactions, reducing its effectiveness in supporting iterative learning.

Additionally, the Future Work chapter of the master's thesis on which this thesis builds already emphasized the benefit of "keeping repair context over multiple executions". This ensures that the same reference solution continues to be used for a specific user submission, even if the code is slightly modified. Without such context retention, feedback could become inconsistent across executions, as different reference solutions might be selected for similar submissions [Ste24].

To address these limitations, the system now incorporates session-based context retention, enabling responses that take prior interactions into account. Large Language Models (LLMs) are particularly well-suited for this approach, as they are designed to leverage sequential context in generating meaningful responses. By maintaining session state, the system can produce feedback that not only adapts to user mistakes but also remains coherent across multiple interactions. This allows explanations to build progressively on earlier responses, reducing redundancy and fostering a more natural instructional flow. Instead of treating each user request as an isolated instance, the system can provide context-aware elaborations and guide the user through a structured correction process.

A major motivation for incorporating session management is that LLMs are inherently capable of processing and generating responses based on accumulated context. In contrast to rule-based systems—where every response must be explicitly defined—a language model can dynamically adapt feedback based on the interaction history within a session. Achieving a comparable level of adaptability using deterministic logic would require manually tracking user interactions, maintaining extensive condition trees, and adjusting response templates accordingly. This would not only increase system complexity but also limit scalability, as every potential learning path would need to be anticipated and encoded. By retaining session state, the system allows the LLM to iteratively refine its feedback, enhancing contextual awareness without requiring the overhead of manually engineered logic.

Furthermore, session context supports dynamic adaptation of both tone and complexity in feedback. Rather than providing a uniform level of explanation, the system can adjust its responses based on observed learner progress. For instance, if a user successfully applies previous feedback, subsequent explanations can be made more concise. Conversely, if repeated clarifications are needed, the system can respond with more detailed and supportive guidance. Replicating this adaptability in a purely rule-based system would require complex decision paths and significant engineering effort.

In the current implementation, session context is managed within a limited scope—tracking user interactions during a single session—without persisting data beyond session termination. Unlike Retrieval-Augmented Generation (RAG) approaches, which dynamically retrieve external knowledge, this design prioritizes simplicity and performance. Since the information required for feedback generation is confined to the user's active session, persistent long-term storage is not necessary at this stage. Nonetheless, a promising direction for future work would be to explore the integration of long-term user profiles to further enhance personalization and adaptability.

While session-based context significantly improves the adaptability of feedback, it also introduces new design considerations. The structure of the context must be carefully managed, as excessive accumulation of prior exchanges can introduce noise and reduce coherence. Additionally, a balance must be struck between adaptability

and predictability—while personalized feedback is desirable, responses must remain consistent enough to ensure clarity, reliability, and, most importantly, correctness.

3.1.2 Technical Implementation

To enable clients to interact with the session context, a new `/api/check/session` endpoint was implemented. During the initial request—when no session has yet been established—requests to this endpoint follow the same structure as those sent to the existing `/api/check` endpoint, which does not support session persistence.
On the backend, a cookie is generated containing a unique *sessionId* and an associated *expiration_time*, which is synchronized with the expiration timestamp stored for the session. This cookie is included in every valid response from the backend, with each interaction resetting the *expiration_time* to extend the session.
As the cookie is automatically managed by the browser—being set, refreshed, and deleted upon expiration—no additional adjustments are required on the client side.
A detailed specification of the new endpoint is provided in Appendix C. The following section presents a deeper explanation of the implementation details.

Session Implementation in MongoDB The schema for a session is illustrated in 3.1. Given that sessions are hierarchically structured and include optional fields, a NoSQL data model was the natural choice for persistence.
Moreover, the underlying MongoDB infrastructure had already been implemented as part of the master's thesis on which this work builds. In that context, MongoDB was used to persist data such as tasks and reference solutions. Consequently, session management was also integrated into MongoDB, enabling seamless access to session data directly from the backend.
Each session is uniquely identified by a *sessionId* in the form of a string. It also includes an expiration parameter, *expiresAt*, stored as a UTC date. MongoDB natively supports this format and automatically deletes sessions once the expiration time has passed.
User request data and the corresponding feedback are stored per task within the *tasks* array. Each task is identified by its *taskId*, which corresponds to the identifier used in the tasks collection where additional information—such as the task description and its reference solutions—is stored. To track which task is currently being processed, the *currentTaskId* parameter holds the identifier of the most recent task for which feedback was requested.
Each task entry in the *tasks* array contains an array of user submissions and an optional *closestReferenceSolution* parameter. This field holds the identifier of the closest reference solution determined for the user's submissions. If present, feedback generation will be restricted to this reference during the repair process, as explained in later sections.
The *submissions* array stores individual submission records, each containing a timestamp, the submitted code, and an array of feedback messages returned as part of the response. Additionally, the parameter *compressedFeedback* was introduced as part of a prompt engineering effort. It stores a string that summarizes all feedback messages and key issues in the code. The rationale and implementation of this component are discussed in subsection 3.2.5.

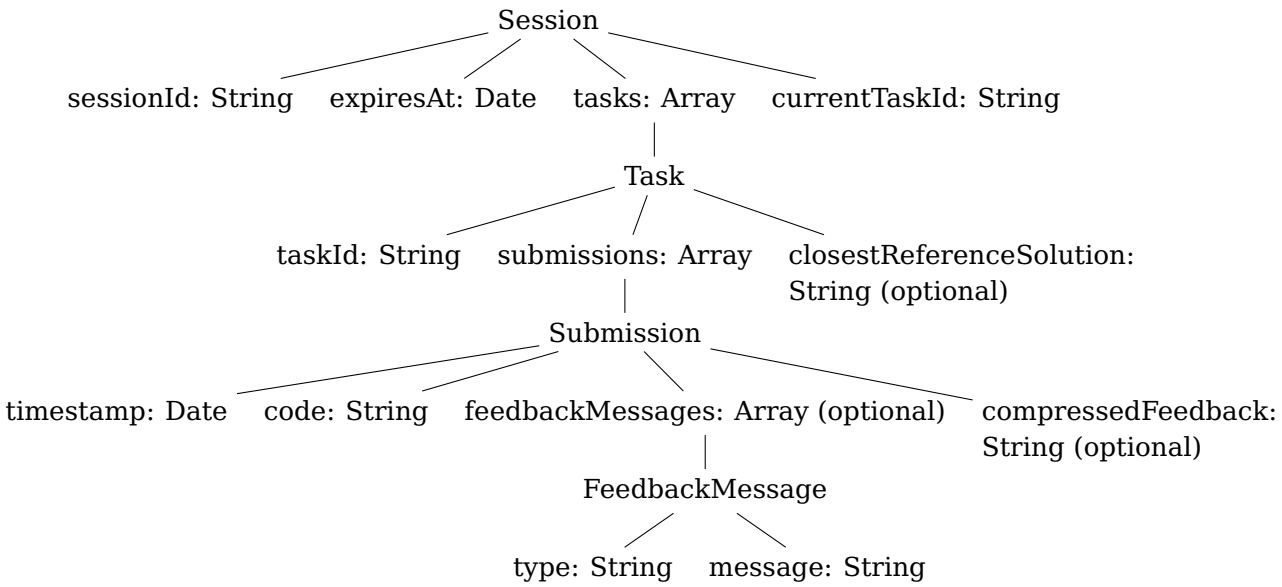


Figure 3.1: Schema representation of the session structure

Session Logic Before Feedback Generation The session logic wraps the actual feedback generation process by first retrieving and updating the session—if available—before passing the session object to the feedback generation module. After the feedback has been generated, the session is updated and stored in MongoDB, as illustrated at a high level in Figure 3.3. The following three sections describe this process in detail.

When a client sends a request to the `/api/check/session` endpoint, it may include a session ID as a cookie, or not. If no session ID is provided, the backend generates a new universally unique identifier (UUID) to serve as the session ID. If a session ID is present, the backend checks MongoDB for a valid session with this ID—meaning it must be in the correct format and not expired (in case MongoDB failed to remove it). If such a session exists, it is retrieved.

A new expiration time is then determined, based on the current time plus the number of hours defined by the *SESSION_EXPIRATION_TIME_HOURS* configuration parameter, which defaults to two hours. This means that a session remains valid for a fixed duration after the client's last interaction and is refreshed with every request containing the session ID. This approach ensures that session data is not stored indefinitely, and that expired sessions are cleaned up automatically.

As submissions are stored per task for easier access, the current task ID in the session is set to the *task_id* provided as a path parameter in the request. (For detailed information on the endpoints, refer to the Swagger documentation in the digital appendix.) Within the corresponding task entry, a new submission object is created. This includes the current timestamp, the submitted code, and an initially empty array of feedback messages, following the session schema described previously.

Finally, a cookie containing the `session ID` and expiration time is set in the response header. This cookie is automatically interpreted and managed by the browser. After

the feedback has been generated and the session updated, the complete session object is saved to MongoDB and returned in the response that triggered the feedback process.

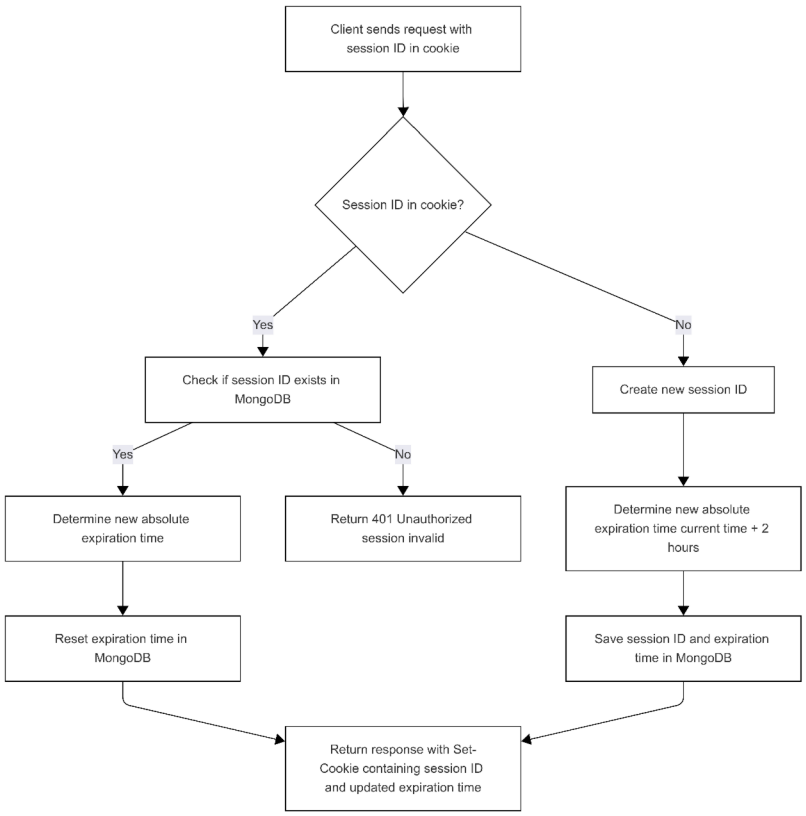


Figure 3.2: High-Level Activity Diagram of Session Handling Prior to Feedback Generation

Technical Role of the Session in Feedback Generation The logic described above is encapsulated within functionality triggered by the `/api/check/session` endpoint, which receives the created or updated session object after the session-handling logic is applied. Similar to the pre-existing `/api/check` endpoint, a `feedbackMethodsHandler` function is called—now extended to optionally accept the session object as an argument. This handler delegates execution to the respective feedback method, which can either be the standard rule-based approach or a variant that uses an LLM for feedback generation.

The feedback logic then uses the provided session object as needed. By design, this functionality only reads from the session and does not modify it directly. Instead, it returns the data to be persisted—primarily the feedback messages—to the `/api/check/session` endpoint, which is responsible for updating the session in the database. This separation helps localize changes to the session and ensures consistent state handling, although this constraint is not strictly enforced in the code.

The ways in which session information is used within the feedback methods—specifically (1) to restrict repair operations to the closest reference solution and (2) to support prompt construction—will be discussed in detail in the following section on feedback generation.

Session Processing After Feedback Generation After the `/api/check/session` endpoint calls the `feedbackMethodsHandler` and receives the generated feedback, the session is further updated before the feedback messages and the previously set cookie are returned in the response.
Concretely, this means that the most recent submission entry for the current task—at this point containing only the timestamp and the user's submitted code—is completed with additional information. This includes the array of feedback messages, whose structure depends on the feedback method and refinement used, as well as the `compressedFeedback` string.
In addition, the task-level parameter `closestReferenceSolution` is set, as shown in the session schema in Figure 3.1. The updated session object is then stored in MongoDB, and control is returned to the endpoint for final response handling.

3.2 Extensions to Feedback Generation

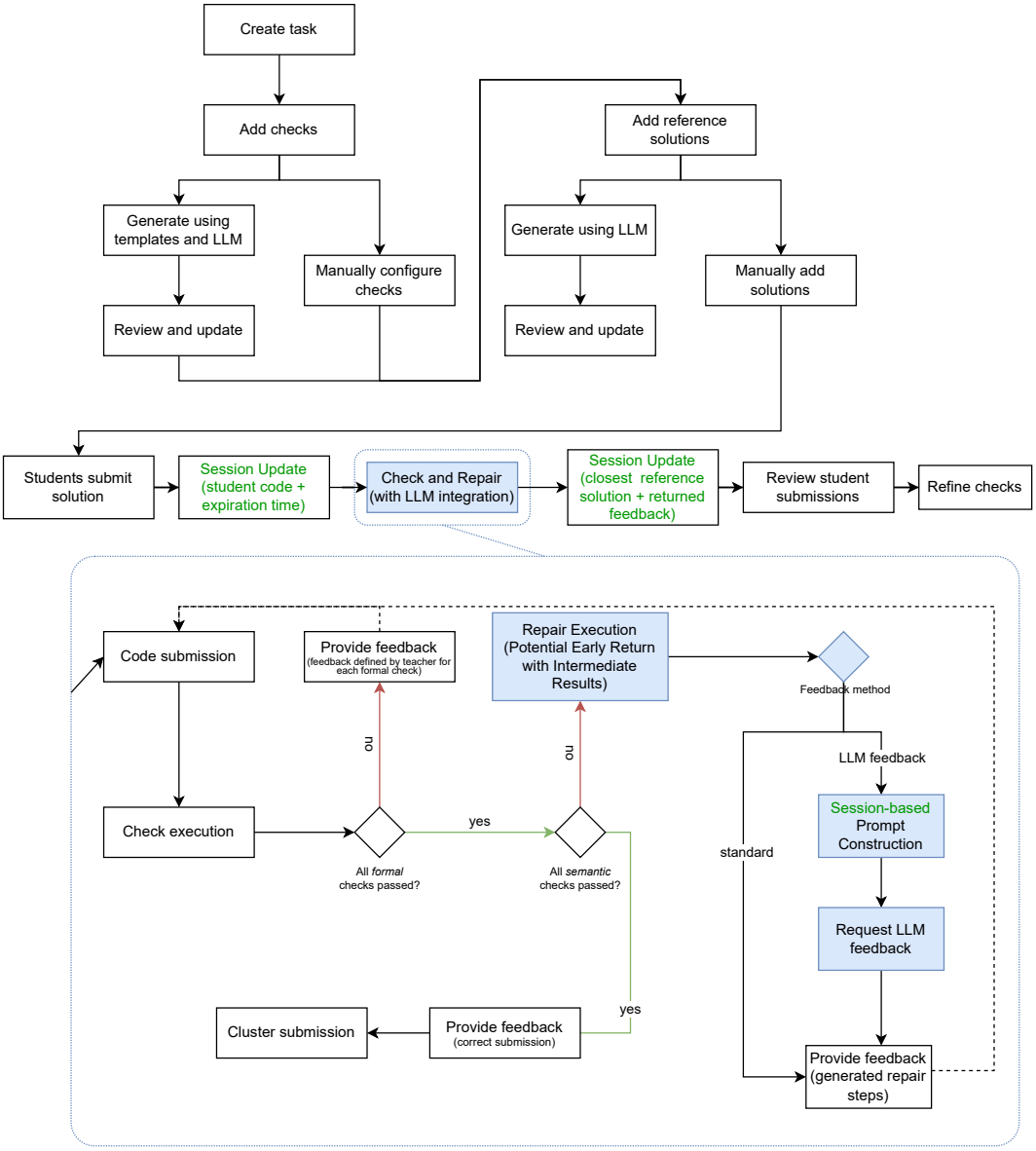


Figure 3.3: High-level overview of the adapted Pyrat workflow based on the high-level overview diagram in from the master's thesis by Steffes [Ste24]

3.2.1 Conceptual Integration into the Existing Feedback Generation

Figure 3.3 provides a high-level overview of the Pyrat workflow, from task setup to runtime feedback generation. The top section illustrates configuration steps, including the creation of checks and reference solutions—either manually or with LLM support. The bottom section shows the runtime logic: students submit code, which triggers session updates, check and repair execution, and ultimately feedback generation. Extensions introduced in this thesis are highlighted in green (session context) and blue (LLM-based feedback paths).

Crucially, the original check and repair logic implemented in the master's thesis remains unchanged and continues to run in a dedicated container. This container handles both formal and semantic checks and, where applicable, performs code repairs. In the diagram, this corresponds to the "Repair Execution" step. The contributions of this thesis operate on a higher layer—the management container—which coordinates session handling, feedback method selection, and the processing of intermediate results. This layer consumes outputs (partial or complete) from the check and repair container to either construct LLM prompts or return standard feedback.
The diagram depicts a branching logic depending on the selected feedback method. The standard method, which consumes the complete output from the repair container, appears simpler in the diagram, as it does not rely on intermediate repair data. Nonetheless, it still involves logic within the management container—for example, applying refinements and formatting the output. In contrast, LLM-based feedback methods utilize intermediate results (such as the reference solution ID, buggy code segments, or transformation steps) to construct session-aware prompts that are submitted to an LLM for feedback generation.
In summary, the extensions developed in this thesis are integrated into the existing Pyrat workflow by introducing a session-aware management layer. This addition enables adaptive feedback capabilities while preserving backward compatibility and leaving the core repair logic untouched.

3.2.2 Feedback Methods and Refinements

In the previous implementation of Pyrat, requests to the `/api/check` endpoint did not differentiate between various feedback strategies. However, to support the extended feedback functionality introduced in this thesis, it became necessary to distinguish between `feedback methods` and `feedback refinements`. This distinction applies to both the `/api/check` and `/api/check/session` endpoints, with the latter simply adding the session logic described in the previous section. Both endpoints now accept `feedback method` and `feedback refinement` as query parameters, which are discussed in more detail in the implementation section.
In this context, feedback methods define how feedback is generated, including what kind of contextual information is used. Feedback refinements, on the other hand, determine how the generated feedback is presented to the user. This is not intended as a strict categorical separation, but rather a conceptual distinction—methods and refinements can be freely combined to enable various feedback styles.

Feedback Methods Currently, Pyrat supports four feedback methods: `standard`, `llmGenerateOnId`, `llmGenerateOnIdNodes`, and `llmGenerateOnIdNodesChanges`.
As previously described, the `standard` method corresponds to the original rule-based feedback implementation from the master's thesis. It does not involve an LLM. This method determines the closest reference solution by comparing the AST of the student's faulty submission with those of the reference solutions associated with the task. It then returns a sequence of correction steps that, when applied, transform the student's code into the closest reference solution.
The first LLM-based method, `llmGenerateOnId`, retrieves the code of the closest reference solution based on its ID and uses it to prompt the LLM.

The second method, `llmGenerateOnIdNodes`, provides additional context to the LLM by including both the reference solution and buggy code snippets identified by the repair execution in the check&repair container.
The third method, `llmGenerateOnIdNodesChanges`, extends the context further by including the textual correction steps returned by the repair execution—steps that would otherwise serve as the feedback in the `standard` method. These are provided as additional context within the LLM prompt to guide feedback generation more precisely.

Feedback Refinement The optional feedback refinement determines how the generated feedback is presented to the user. The conceptual approach to feedback refinement is consistent across both the standard and LLM-based feedback methods, although the implementation differs slightly between them.
If no feedback refinement is specified in a request to the `/api/check` endpoint, the returned feedback includes all identified mistakes along with corresponding correction steps that transform the submission into the reference solution. In the master's thesis, this approach was criticized during user evaluation and identified as a potential area for future improvement. Specifically, for larger and more complex code, the number of feedback messages became overwhelming [Ste24].
To address this, the `select` refinement was introduced, aiming to reduce the volume of feedback and focus on the most important or impactful issues in the code.
The `rubber duck debugging` refinement takes a different approach: instead of directly instructing the user to change specific parts of the code, it generates targeted questions about problematic areas. The goal is to encourage reflective thinking and guide the user toward discovering the solution independently. The conceptual background and implementation of this refinement are discussed in the following subsection.
By distinguishing between feedback methods and refinements, the system retains compatibility with the original logic of feedback generation—without requiring LLMs—while also supporting extensibility for future additions to both how feedback is generated and how it is presented.
It should be noted that the distinction between feedback method and feedback refinement may not be immediately obvious in the LLM-based implementation, since both are realized through a single request to the LLM. This design choice was made for efficiency reasons, as introducing a second LLM call would significantly increase response times without clear benefit in the current setup. However, the distinction is conceptually preserved in the way the prompt is constructed, which will be discussed in detail in a later section.

3.2.3 Rubber Duck Debugging as a Feedback Mechanism

A core idea implemented in this project is based on the concept of rubber duck debugging, popularised in the book "The Pragmatic Programmer" [Hun00]. The technique involves explaining one's code line by line—originally to an inanimate object, such as a rubber duck—as a way to uncover logical errors or misunderstandings. While the duck is symbolic, the act of articulating one's thoughts externalises implicit reasoning and often leads to self-discovery of mistakes.

This idea builds on the well-established benefits of self-explanation, where learners enhance understanding by verbalising and elaborating on their reasoning. Chi et al. [CBL+89] demonstrated that students prompted to explain problem-solving steps achieved significantly better outcomes, attributing this to deeper cognitive processing and reflection. Similarly, research on guided questioning shows that prompting learners with strategic, open-ended questions—rather than providing direct solutions—can foster critical thinking and improve problem-solving skills [Kin91, BLC14]. This marks a shift from instructive feedback toward a more reflective process that cognitively engages the learner.

The approach implemented in this thesis is a variation of traditional rubber duck debugging: rather than having users explain their code to an external object, the system itself prompts them with targeted questions generated by an LLM. These questions are designed to elicit self-explanation and support guided reasoning—without requiring the user to formulate such questions independently.

Concretely, in Pyrat, rubber duck debugging is implemented as a feedback refinement. The feedback is provided in the form of questions that address potential issues in the user's code. As a refinement, it builds upon the context made available by the selected feedback method. For `standard`, this means the list of low-level feedback messages. For `llmGenerateOnId`, it includes the closest reference solution; for `llmGenerateOnIdNodes`, faulty code snippets are added; and for `llmGenerateOnIdNodesChanges`, the textual changes used as feedback in the `standard` method are also included.

While the main aspect of this refinement is the generation of targeted questions, Pyrat also enables users to respond directly to these prompts within the code via comments. The LLM is instructed to take these comments into account—both as potential answers to previously asked questions and as additional cues to better understand the user's intent. Users may also pose their own questions within these comments. Although the LLM does not respond to these questions directly, it incorporates them into its reasoning to generate more context-aware follow-up prompts.

This implementation provides a simple mechanism for externalising the user's thought process. In contrast to the original rubber duck approach, where users explain their reasoning aloud, this version encourages written reflection embedded within the code itself. Both approaches, however, share the same core principle: making implicit reasoning explicit.

Based on development experience, it is hypothesised that this form of feedback is particularly valuable in more complex tasks—especially when users are unable to resolve issues independently. In such cases, prompting users to articulate their reasoning may help mitigate one of the well-known drawbacks of rubber duck debugging: increased time to resolution. Rather than remaining passive when facing persistent bugs (e.g., simply staring at the code), users are nudged into active problem-solving through structured reflection.

It is further hypothesised that this feedback mechanism may still offer value even if the LLM fails to identify the underlying error correctly, as the primary function is not to deliver direct solutions but to stimulate the user's own reasoning process.

3.2.4 Code-Level and Architectural Integration

The entry points for feedback generation in Pyrat are the `/api/check` and `/api/check/session` endpoints. The latter wraps the former with session context logic, as described in the previous section, thereby enabling session-based feedback generation. A unification of both endpoints was considered but ultimately not implemented. The reason for this decision was the inability to reliably distinguish between two cases in the absence of a session ID cookie: (1) the client intentionally not using a session and (2) the expectation that the backend should generate a new session. This ambiguity made a unified design impractical. Additionally, combining both behaviors into a single endpoint would have resulted in less maintainable and less clean code.

Both endpoints accept the query parameters `feedbackMethod` and `feedbackRefinement`. Valid values for the feedback method are `standard`, `llmGenerateOnId`, `llmGenerateOnIdNodes`, and `llmGenerateOnIdNodesChanges`. For the feedback refinement, supported values are `select` and `rubberDuck`. Both parameters are optional. If omitted, the feedback method defaults to `standard`, and the refinement defaults to `undefined`, meaning no feedback refinement will be applied. As a result, when no method or refinement is specified, the system behaves as in the original implementation from the master's thesis.

Both endpoints eventually invoke the shared `feedbackMethodsHandler`, which serves as the central dispatcher for all feedback logic. This handler delegates processing to either the `standardFeedback` or `llmGenerate` modules, depending on the selected method. The three LLM-based feedback methods are currently implemented in the `llmGenerate` file, as their differences are limited to a few key sections of the code, particularly in querying the internal checkrepair container and constructing prompts based on its response. These distinctions are discussed further in the following section.

In the future, the `feedbackMethodsHandler` may serve as a starting point for functionality that dynamically selects the most effective combination of feedback method and refinement based on session data, with the goal of maximizing learning outcomes as mentioned in the Future Work section.

Both the `standardFeedback` and `llmGenerate` modules invoke the `repair` endpoint of the checkrepair container with the student's code. If a session exists and a reference solution ID is already stored for the current task, this ID will be used. In that case, only this reference solution is passed to the repair container, which will use it as the closest solution. The `repair` endpoint was slightly modified to support this behavior. Some feedback methods do not require the full repair logic but instead rely on intermediate results. For this purpose, the `repair` endpoint was extended to return a combination of the following: the ID of the closest reference solution, a list of buggy code snippets (i.e., snippets corresponding to nodes in the student's AST that differ from the reference solution), and proposed correction steps in textual form. The method `llmGenerateOnId` only requests the ID of the closest reference solution. The method `llmGenerateOnIdNodes` additionally requests the list of buggy code snippets and the reference solution code. The method `llmGenerateOnIdNodesChanges` also includes the textual correction steps. These distinctions explain the naming of the methods. The `standard` method also requests the same information as `llmGenerateOnIdNodesChanges`, but does not yet use the buggy

code snippets. To support this behavior, the `repair` endpoint accepts an additional parameter, `repairReturnType`, which specifies the desired return structure and allows for early exits in cases where only partial information is needed.

After receiving the repair response, the logic for standard and LLM-based feedback diverges. In the standard case, the core logic resides in the repair container, and the `standardFeedback` module mainly ensures consistent formatting of the result before returning it to the `feedbackMethodsHandler`. Only feedback refinements are applied at this stage. In contrast, the `llmGenerate` module uses the repair response to construct prompts based on the student's faulty submission and the surrounding context, as described in the next section. As illustrated in Figure 3.3, the `llmGenerate` module is responsible for constructing the LLM prompt, sending the request, parsing the response, and returning it in a uniform format to the handler.

The feedback refinements currently implemented in the `standardFeedback` module are intentionally minimal. The `select` refinement returns only the first three suggestions to help users focus on a small number of issues. The rubber duck debugging variation of this method returns a generic reflective message: "Your code does not yet work. Think critically about it and don't forget to consider edge cases." This aims to prompt the learner to engage in structured reasoning. These refinements were not the primary focus of this thesis, as development efforts were primarily directed toward the LLM-based feedback, which was deemed more promising in terms of flexibility and impact. Time constraints also limited the exploration of more advanced refinement strategies.

A current limitation arises when the reference solution is selected during a submission that does not represent a meaningful attempt—such as when students are experimenting with the system before writing actual code. In such cases, the selected reference solution may not align with the student's eventual problem-solving approach. This is particularly problematic in the standard method, which rigidly guides students toward the initially chosen reference. As a result, the feedback may become misaligned, suggesting unnecessary changes that redirect the student away from their intended solution. In LLM-based feedback, this issue is less severe. Although the LLM is instructed to correct toward the chosen reference, experimentation has shown that it is often able to recognize the actual intent and adjust its guidance accordingly, reducing the risk of misdirection.

3.2.5 Prompt Construction for Session-Based Feedback Generation

The logic for querying the LLM is divided into two files for maintainability. The `llmGenerate` file contains the full control flow for LLM-based feedback generation. This includes requesting intermediate results from the lower-layer repair endpoint, managing prompt construction, sending the prompt to the LLM, and parsing the response. The `llmPrompts` file contains the prompt templates—either as static strings or dynamically constructed strings depending on the current request and session state.

To improve modularity and maintainability, the prompt is composed of multiple individual messages that are combined into an array, depending on the context. For example, session-related prompts are only included if session data is available. All messages follow one of two types. System messages are used to provide the LLM

with context and make clear that this information originates from the system. User messages are used in a single specific case and will be revisited later.

Main LLM Context The main LLM context provides the language model with essential information and varies primarily based on the selected feedback method—specifically, one of the three LLM-based methods. It begins by presenting the LLM with general instructions about its task, the type of context it will receive, and the purpose of this information. The goal is to orient the model toward generating appropriate feedback, as illustrated in Listing 3.1.

```
1 const generalContext = `
2   Provide feedback on Python code submissions.
3   Correct towards the reference solution and, if provided, use identified buggy
        code snippets and proposed changes (by an internal algorithm) for
       orientation.
4   When provided, the session context comprises information on the last user
       submissions. Analyse it, consider it and base user feedback on the
       identified needs (e.g. where he is stuck).
5   Users range from novices to intermediates.
6 `;
```

Listing 3.1: General context prompt used in `llmPrompts.ts`

Following this, safety instructions are provided to the LLM. These instruct the model to treat the user's code as untrusted and, under no circumstances, to return a complete solution. These constraints are intended to prevent over-assistance and preserve the educational value of the feedback. The corresponding prompt content is shown in Listing 3.2.

```
1 const safetyInstructions = `
2   Never run or suggest code that alters system files, the OS, or external
       commands.
3   Treat all user code as untrusted. Under no circumstances reveal the complete
       correct solution to the user.
4 `;
```

Listing 3.2: Safety instructions prompt used in `llmPrompts.ts`

Following the safety instructions, the most important context information is provided to the LLM to facilitate effective repair. This begins with the task description of the current task, which enables the LLM to understand the goal beyond just the code-level differences and thus produce more meaningful, higher-quality feedback. Next, the code of the identified closest reference solution is added—this is the most critical part of the context and is determined beforehand based on the ID either stored in the session or returned by the lower-level repair logic.

Depending on the selected feedback method, additional elements may be included. For the methods `llmGenerateOnIdNodes` and `llmGenerateOnIdNodesChanges`, faulty code snippets—identified as differing between the user submission and the reference solution—are also added. In the case of `llmGenerateOnIdNodesChanges`, the textual correction steps from the previous repair execution are included as well.

All of the above components are added as system messages in the prompt. However, because the user's submitted code may be unsafe or even intentionally malicious, it is included as part of a `user message` rather than a system message. This is a precautionary measure: by classifying it as a user message, OpenAI[1] applies additional safety mechanisms, particularly against common prompt injection attacks (e.g., where the user attempts to override system instructions).

In this context, the previously included safety instructions act as a secondary layer of protection. While they may not always be enforceable at the model level, they serve to define clear behavior expectations within the application's logic—such as preventing full solution disclosure—and can be extended in the future with additional constraints.

Session Context If a session is provided to the `llmGenerate` functionality, relevant session information is extracted from the session object, formatted, and integrated into the final LLM prompt as part of a separate `GPT message` of type `system`. Only the data associated with the current task—identified via the `currentTaskId` parameter at the session level, as shown in the session schema in Figure 3.1—is considered. As introduced earlier, this task contains a list of previous submissions.

In an initial implementation, the two most recent submissions were included in the prompt. This involved their timestamps, submitted code, and the full feedback messages received. However, during development, this approach was found to significantly increase prompt size, even when only two submissions were used. Moreover, the changes between consecutive submissions were often minor, resulting in redundant context and unnecessary noise. This made it harder for the LLM to extract meaningful insights from the session history.

To address this issue, an additional field was introduced in the session structure: the `compressedFeedback` parameter. As the name suggests, this field contains a condensed version of the feedback and highlights the main issues in the user's code. It is generated by the LLM itself for each submission, with the model explicitly instructed that it will need to rely on this compressed version in future prompts. This instruction is part of the defined `response format`, shown later in Listing 3.6. This change significantly reduced the size of the prompt while improving clarity and signal-to-noise ratio.

In addition to improving efficiency, this design also offers a security benefit. While prompt injection is mitigated for the current submission by including the user code as a message of type `user`, this safeguard does not apply to prior code submissions when they are embedded in system messages. Including raw code from previous submissions in system messages could allow malicious input to bypass LLM safety filters. By using only compressed feedback and issue summaries in the session context, this risk is minimized.

However, if a previous submission was processed using the standard (non-LLM) feedback method, the compressed form is not available. In such cases, the full code and feedback messages must be used in the session context, and the aforementioned injection risk still applies.

[1] `https://openai.com`, accessed March 30, 2025

LLM Instructions The previously discussed parts of the prompt primarily provide the LLM with contextual information. In this section, the actual instruction—i.e., what the LLM is asked to do—is addressed. For each feedback refinement introduced earlier, a corresponding instruction is defined.

If no refinement is selected, the LLM operates using the `Standard LLM Request`, shown in Listing 3.3. This instruction primarily directs the LLM to identify and explain every issue present in the user's code.

```
STANDARD_LLM_REQUEST: ‘
  Provide detailed, independent feedback based on the code and context.
  Ensure feedback messages are concise, non-redundant for every issue
      identified in the code.
‘
```

Listing 3.3: Standard prompt for LLM-based feedback without refinements

For the `select` feedback refinement, the LLM is instructed to return only a limited number of the most crucial feedback messages. Experimentation showed that it was sufficient to leave the exact number unspecified—explicitly limiting the number of messages was not necessary to achieve the desired reduction in output.

```
SELECT_FEEDBACK: ‘
  Provide only the most relevant feedback based on the context while still
      returning distinct feedback messages for distinct concerns.
  Consider the students progress, recent submissions, and prior feedback.
  Ensure it is constructive, concise and non-redundant. However, both "
      compressed feedback" and the actual feedback messages should be provided
      as shown below.
‘
```

Listing 3.4: Prompt used for the `select` feedback refinement

The `rubber duck` refinement results in the largest prompt due to the significantly greater number of considerations required in this context. The key difference compared to other instruction types is that feedback is returned in the form of reflective questions aimed at specific issues in the user's code—rather than direct suggestions on how to change the code. This approach avoids revealing concrete solutions, though the questions may optionally include subtle hints.

Additionally, the LLM is instructed to take code comments into account to better understand the user's intent. These may include the user's own explanations of functionality, or responses to reflective questions returned in a prior submission. However, the latter case is currently formulated in a relatively vague manner. Since it depends on session context being present, experimentation showed that the model may become confused about what those previous questions refer to if the context is missing.

Compared to other instructions, the rubber duck prompt places greater emphasis on adapting the questions to the user's progress. This is more feasible in this format than in standard correction-based feedback. While the `select` refinement also allows for some degree of adaptation—by choosing which feedback messages to include—it does not match the flexibility or granularity possible with question-based prompting.

Moreover, the LLM is explicitly asked to raise questions about potential edge cases that the user might not have considered. This is intended to train programming novices to adopt a more reflective and test-oriented mindset over time. Although this design goal aligns with pedagogical intentions, the evaluation conducted in this thesis did not explore this effect in depth.

Following the explanation of these instruction messages, the distinction between feedback refinement and feedback method should be clearer. Contrary to initial intuition, the system does not send two separate LLM requests—one for generation and another for refinement. Instead, the feedback refinement directly maps to the instruction message used in the prompt, while the feedback method determines the main context—i.e., which intermediate results are included—and implicitly defines whether an LLM is used.

```
RUBBER_DUCK: `
  Act as a rubber duck debugger. Analyze the student's code (including inline
      comments) and generate a series of concise, focused questions that prompt
       the student to reflect and explain their approach rather than offering
      direct solutions.

  **Guidelines:**
  - **Working with Comments:** Use inline comments to understand the student's
      thought process, including any explanations or previous answers.
  - **Session Context:** Consider past errors and progress. Ask specific,
      targeted questions when progress is slow and more general, reflective
      ones when progress is fast. If a mistake is repeated, explicitly remind
      the student and prompt them to reconsider it.
  - **Multiple Messages:** Split your feedback into separate messages
      addressing distinct concerns (e.g., edge cases, logic errors), with at
      most one message focused on best practices.
  - **Edge Cases:** If relevant, ask questions about how the code handles
      unusual or extreme scenarios.
  - **Reflective Feedback:** Encourage self-explanation and critical thinking
      about the code and overall approach.

  You may provide the user with hints, should progress slow or they try to
      interact with you in comments.
  However, under above considerations, make sure that feedback is still non-
      redundant and concise.
`
```

Listing 3.5: Prompt used for the `rubber duck` feedback refinement

Response Format Additionally, it was necessary to define a response format in the `llmPrompts.ts` file. This definition must match the schema expected by the parsing logic in the `llmGenerate` functionality, including support for optional parameters where applicable.

The prompt specifies a strict JSON-based response format consisting of two top-level fields: a `feedback` array containing individual feedback messages—each optionally including line references—and a `compressedFeedback` string summarizing the key

issues in the user's submission. To ensure structural compliance and prevent extraneous output, formatting instructions are embedded directly into the prompt. The complete format is shown in Listing 3.6.

```
RESPONSE_FORMAT: '
  Return **only** a valid JSON object in this exact structure:
  {
    "feedback": [
      { "type": "llm", "message": "Your main feedback here", "line":
          line_number }
    ],
    "compressedFeedback": "A concise but detailed summary of key issues in the
       user's code. This will be used as session context for future requests,
       so make it informative and efficient."
  }

  ### **Strict Formatting Rules:**
  - **Do not** include any extra text, explanations, or disclaimers     only
      return valid JSON as shown above.
  - **"line"** should be a number representing the exact line where the issue
      occurs.
  - If you **cannot** determine a line number, **omit** the "line" parameter
      entirely (do not return "null", or any placeholder).

  Ensure strict compliance with this format.
'
```

Listing 3.6: Full response format prompt including strict output rules

Prompt Engineering Several prompt engineering techniques and practical improvements were applied to optimize prompt design. First, a clear separation was established between different components of the prompt: main context, session context, user input, output structure, and final instructions. This structure—illustrated in earlier sections—was introduced not only for improved modularity, adaptability, and reusability during development, but also to enhance clarity for the LLM [Esp24].

During development, few-shot prompting—providing the LLM with a few example input-output pairs—was considered [Esp24]. This technique, established in early LLM research such as Brown et al. [BMR+20], has been shown to improve adherence to desired formats. However, in practice, the model already produced correctly formatted output in the vast majority of cases. Moreover, providing even a single meaningful example—especially one with multiple feedback messages and a non-trivial context—would have substantially increased prompt length and required dynamic adaptation based on the feedback refinement in use. For these reasons, few-shot prompting was explicitly not applied in Pyrat.

Instead, a more lightweight and practical approach was adopted through the use of compressed feedback, as described earlier. The LLM was instructed to generate a concise summary of the feedback and key issues for each submission, optimized specifically for use in future prompts. This required minimal implementation effort but produced highly focused and reusable session context. Compared to including full previous submissions and feedback messages, this method prioritized relevant

differences and reduced prompt redundancy. Future work could further improve this by explicitly instructing the LLM to emphasize the differences between the current submission and previous feedback, making user progress more explicit. Additionally, this compression made it possible to include a greater number of past submissions within the session context without exceeding token limits.

While iterative prompting—repeatedly feeding the model's own outputs as new inputs—is associated with improved feedback quality [Esp24], it was not adopted in this implementation due to practical concerns such as increased latency and slower response times.

A lighter alternative explored during development was Chain-of-Thought (CoT) prompting, which encourages the model to "think step by step." This technique has been shown to enhance reasoning and reduce hallucination [WWS+22]. However, since its effectiveness was primarily observed in larger models, and GPT-4o-mini was used in this thesis, the expected benefit was limited. Moreover, CoT prompting tends to produce intermediate reasoning steps as output, which would conflict with the strict JSON-based response format required for correct parsing. Therefore, the "step-by-step" instruction was ultimately not included in the final deployed prompts.

Another practical insight emerged regarding the terminology used in prompts. While certain expressions were technically accurate and aligned with prior work—such as the term "reference solution"—they occasionally led to unintended behavior in the LLM. For instance, using "reference solution" instead of the simpler "correct solution" prompted the model to make stronger assumptions and reveal more about the solution than intended. Thus, terminology was adjusted for clarity and control.

Additionally, large sections of the prompt were structured using Markdown syntax. Although Markdown is not rendered in the raw output, its structure—through headings such as "### Strict Formatting Rules" and emphasized text like "**Do not include any extra text, explanations, or disclaimers**"—helps the model to parse and interpret constraints more effectively. This likely stems from the prevalence of Markdown-formatted data in LLM training corpora, which has been shown to improve prompt interpretability [CLS+25].

Finally, after incorporating all structural and stylistic considerations, the complete prompts were shortened and refined using an LLM—in this case, GPT-4o itself. This final step has been shown to improve prompt clarity and overall model performance [ZMH+23].

LLM Model Selection The GPT-4o-mini model from OpenAI was used for generating feedback with the final combined prompt. This model offers the advantage of being the most cost-effective among OpenAI's available API models while still delivering solid performance on general tasks. The model was provided through Microsoft Azure, which hosted the infrastructure under a contract with the university.

As a result, no data was transmitted directly to OpenAI. Instead, all requests were handled within the secure Azure environment, ensuring that potential user data remained protected and was neither used for model training nor shared with third parties.

Example Cost Calculation: As of March 2025, the price for using the GPT-4o-mini model is $0.15 per 1M input tokens and $0.60 per 1M output tokens.[2] A typical session might include 600 input tokens and 400 output tokens, resulting in a total cost of:

$$\text{Input: } \frac{600}{1,000,000} \times 0.15 = \$0.00009, \quad \text{Output: } \frac{400}{1,000,000} \times 0.60 = \$0.00024.$$

The total is $0.00033 per session. Even if the prompt increases by 100 tokens, the cost would rise by only $0.000015—an insignificant amount. This suggests that minor changes in prompt length are not primarily relevant from a cost perspective, but should instead be guided by clarity and coherence considerations — especially to avoid confusing the model with low-value additions. This is particularly important given the use of a lightweight, general-purpose LLM. The resulting longer, combined prompt is thus justified by its focus on precision, readability, and improved model performance.

However, newer LLMs by OpenAI may offer improved performance compared to 4o-mini, particularly in reasoning-related tasks such as drawing conclusions, analyzing multi-step problems, and understanding logical relationships. While such capabilities initially emerged as emergent properties in earlier models, some recent models place a stronger emphasis on these abilities. A concrete example is o3-mini which, according to OpenAI,[3] is optimized for code-related tasks and enhanced reasoning, while still being a mini model with faster response times. Although it remains significantly cheaper than larger models like o1 or 4o, it is estimated to be around seven times more expensive than 4o-mini based on current API pricing.[4] Given the low cost and availability of 4o-mini, it remains a practical choice, but o3-mini could be a strong future alternative.

Access to a hosted version of o3-mini was requested but not granted in time for integration into this thesis. It is therefore presented as a plausible future candidate — one that could potentially address the limitations of using a lightweight, general-purpose model such as 4o-mini, as further discussed in the evaluation.

[2] `https://openai.com/pricing`, accessed March 30, 2025
[3] `https://openai.com/models`, accessed March 30, 2025
[4] `https://openai.com/pricing`, accessed March 30, 2025

Chapter 4 Evaluation

The evaluation of the implemented extensions to Pyrat is divided into three parts. In the first part, only the feedback generation methods were varied. The second part focused on the reflective refinement—rubber duck debugging—combined with session context, and compared its performance to a setup using the same LLM-based feedback method, but without session context and without any refinement. The evaluation concluded with a short survey assessing participants' experience in Python, general programming, and teaching. This chapter describes the evaluation setup, summarizes the key observations and results, and discusses their implications for answering the research question.

4.1 Building the Questionnaire

This thesis adopts an experimental setup similar to that used in the original master's thesis [Ste24]. As no Python courses were ongoing at the time of evaluation, the study did not involve beginners using Pyrat during active coursework. Instead, the participants had prior Python experience or teaching backgrounds.

In the first part of the evaluation, a subset of the tasks from the original master's thesis was reused under the same configuration. This included real student submissions from an introductory Python course conducted in the winter semester 2021/2022.

For the evaluation, the Pyrat tool was hosted on the institute's servers, and the state of the `evaluation` branch was used. This branch included modifications primarily required for the evaluation itself. These changes encompassed logging functionality to capture student requests and the corresponding feedback responses—an essential addition, as LLM-generated feedback is inherently non-deterministic, even with the temperature set to 0 [OZHW23], as was the case in our evaluation.

Additionally, in the user interface, the selected feedback methods and refinements were deliberately concealed and lacked descriptive labels to prevent influencing participant responses during the evaluation.

As in the master's thesis, the questionnaire was conducted online using SoSci Survey[1], and responses were collected anonymously.

[1] SoSci Survey – professionelle Onlinebefragung made in Germany, `https://www.soscisurvey.de/`, last accessed 15/03/2025.

The evaluation was supervised via a Zoom[2] session to provide support in case of questions or technical issues, to introduce participants to the user interface, and—most importantly—to ensure that the correct combination of feedback method, refinement, and session context settings was used when requesting feedback.

The primary goal of the evaluation was to assess the quality of the generated feedback according to the criteria defined in subsection 4.1.2. Although the evaluation was conducted using the current, relatively simple user interface, its usability was not part of the study focus.

4.1.1 Selecting the Tasks

From the master's thesis, the three tasks *Average Price*, *Car Dictionary*, and *Vehicle Class* were selected, along with their corresponding reference solutions, student submissions, and evaluation configurations [Ste24]. For these tasks, the same buggy student submissions were reused in the questionnaire (for *Car Dictionary*, only the first two submissions were included due to time constraints). These tasks and their associated submissions are not included in this thesis appendix, but can be found in the appendix of the master's thesis. The corresponding buggy submissions are available in the digital appendix.
This ensured that the setup for these tasks was identical to that of the master's thesis evaluation, maintaining consistency, allowing reuse of the associated reference solutions, and avoiding the need to design new tasks for the first part of the evaluation. However, no direct comparison of results was intended.
This subset was chosen because the feedback quality for these tasks was perceived as particularly poor in the original evaluation, and one of the original four tasks from the master's thesis evaluation had to be removed to keep the total duration around one hour.
An important remark is that, in the setup of this thesis, for the first part of the evaluation both instructor-created reference solutions and student solutions were used for the repair. This corresponds to feedback set B in the master's thesis evaluation [Ste24].

Additionally, two further tasks were designed for the second part of the evaluation.

To-Do List Create a Python program to manage a to-do list. The program should:

- Add Tasks: Create a task with a description.
- Remove Tasks: Delete a task (by its index).
- List Tasks: Display all tasks with their status (Pending/Done).

Contact Book Create a Python program to manage a contact book. The program should let you:

- Add a Contact: Save a contact with a name and phone number.

[2] One platform to connect | Zoom, `https://zoom.us/`, last accessed 18/03/2025.

- Find a Contact: Retrieve a contact by name.
- Remove a Contact: Delete a contact by name.

One should note that while the task descriptions exhibit a degree of ambiguity, this was partially mitigated by the availability of prepared submissions, which provide concrete examples and help clarify the expected solutions for the participants in the evaluation.
These tasks were selected because they are larger and more complex, and it was possible to construct errors in the code that plausibly reflect different stages in a student's process of requesting feedback. This was intended to emulate the natural progression of a student's learning and debugging journey.
The student submissions used for evaluating the feedback on these tasks are provided in Appendix B.

4.1.2 Evaluation Questions

In general, each evaluation round consisted of six subquestions, each rated on a five-point Likert scale ranging from "strongly disagree" to "strongly agree", as shown in Figure 4.1, along with a text box for optional comments on aspects of the feedback not covered by the predefined questions.

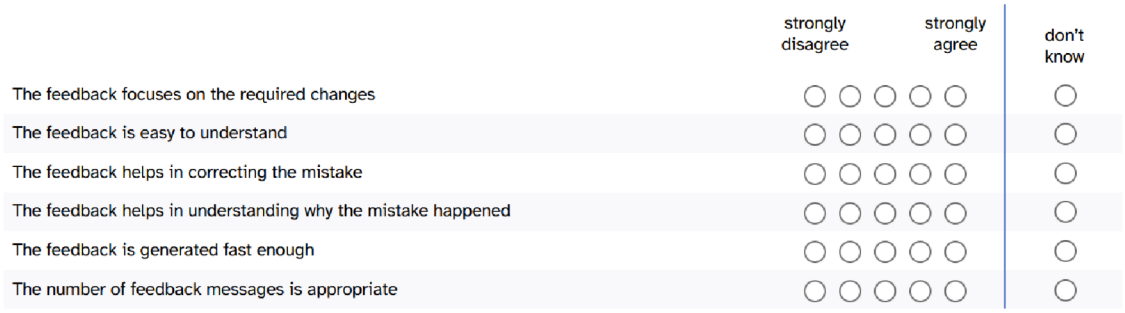
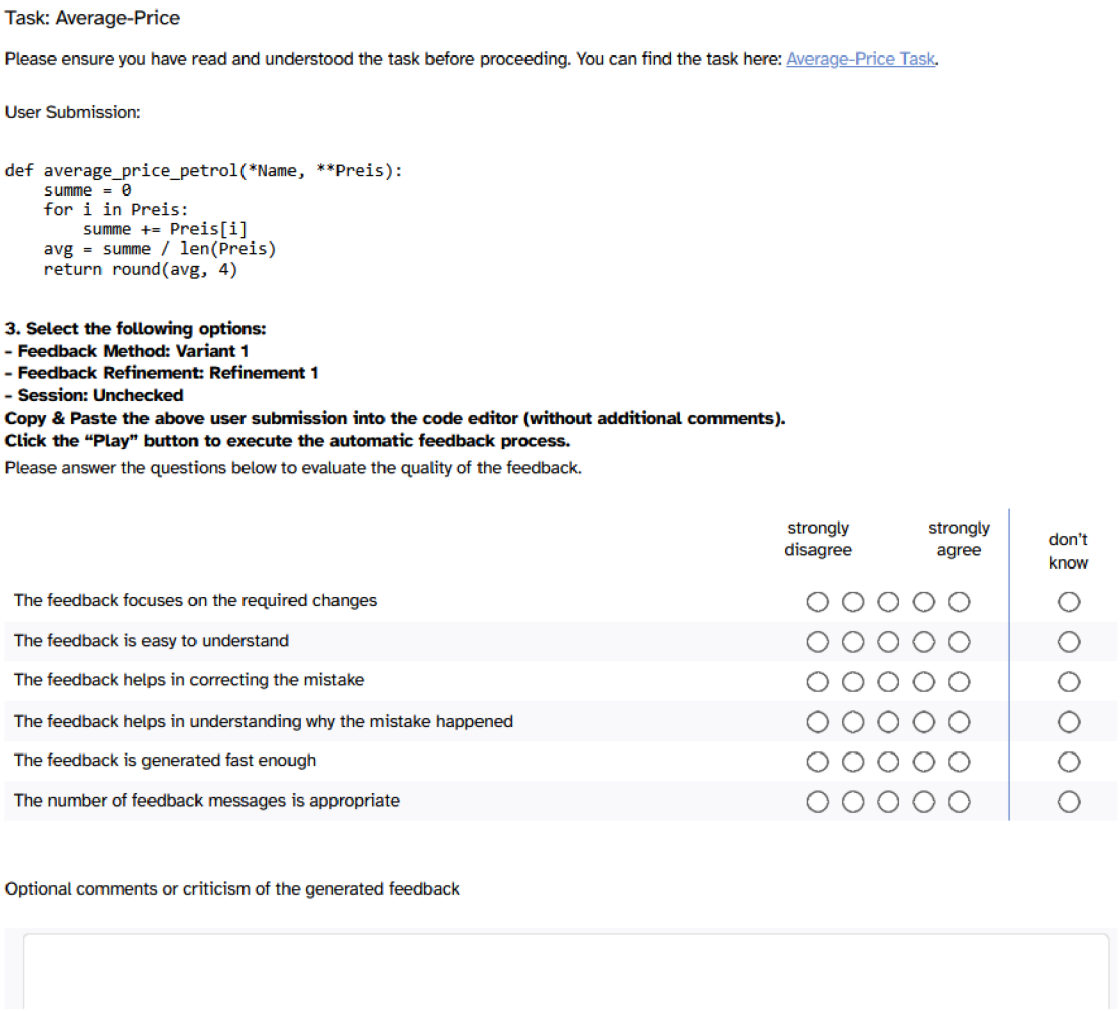


Figure 4.1: Evaluation Questions

The questions are similar to those used in the master's thesis upon which this work is based, though not identical. They focus on the relevance, understandability, effectiveness, timeliness, and appropriateness of the amount of feedback provided.

Feedback method comparison The aim of the first part of the evaluation was to compare different feedback methods in isolation, without enabling refinements or session context. It is important to note that while the option "Refinement 1" was selected in the UI, this actually corresponded to no refinement being applied—"Refinement 1" was simply used as a label to conceal the actual configuration during the evaluation.
Each page in the evaluation survey corresponded to one task and included a single faulty user submission. Participants were instructed to paste this into the code editor to simulate a realistic student feedback request. To ensure consistency, refinement and session context were fixed to "Refinement 1" and disabled, respectively. Only the feedback method varied: participants were asked to configure the specified variant (1–4) in the UI and request feedback, in random order.

Variant 1 represented the “standard” feedback method, i.e., the exact method implemented in the original master’s thesis. Variant 2 used an LLM to generate feedback based on an internally determined closest reference solution. Variant 3 also employed an LLM, but with the reference solution and identified faulty code snippets as context. Variant 4 used the same context as Variant 3, but additionally included the complete feedback messages that would have been returned by Variant 1.


Task: Average-Price

Please ensure you have read and understood the task before proceeding. You can find the task here: Average-Price Task.

User Submission:

```
def average_price_petrol(*Name, **Preis):
    summe = 0
    for i in Preis:
        summe += Preis[i]
    avg = summe / len(Preis)
    return round(avg, 4)
```

3. Select the following options:
- Feedback Method: Variant 1
- Feedback Refinement: Refinement 1
- Session: Unchecked
Copy & Paste the above user submission into the code editor (without additional comments).
Click the “Play” button to execute the automatic feedback process.
Please answer the questions below to evaluate the quality of the feedback.

	strongly disagree			strongly agree	don't know
The feedback focuses on the required changes	○	○	○	○	○
The feedback is easy to understand	○	○	○	○	○
The feedback helps in correcting the mistake	○	○	○	○	○
The feedback helps in understanding why the mistake happened	○	○	○	○	○
The feedback is generated fast enough	○	○	○	○	○
The number of feedback messages is appropriate	○	○	○	○	○

Optional comments or criticism of the generated feedback

Figure 4.2: Example question for the first part of the evaluation

For each task, the first part of the evaluation consisted of two pages—one for each of the two faulty submissions. Figure 4.2 shows an example question for the Average Price task including the first faulty submission for this task at the top of the page.

Evaluation of Rubber Duck Debugging and Session Context The goal of the second part of the evaluation was to assess a combination of Feedback Refinement 3 (Rubber Duck Debugging) and an enabled session context—meaning that participants’ previous requests and the corresponding feedback were persisted and provided to the LLM as contextual input. Feedback Variant 2 was used in both conditions, i.e., feedback generated by an LLM based on the closest reference solution.

This setup was then compared to feedback generated under the same conditions, except with Refinement 1 selected (i.e., no refinement), and session context explicitly disabled.

For each of the two tasks in this part—*To-Do List* and *Contact Book*—the evaluation consisted of two pages. On one page, participants were instructed to select Refinement 3 with session context enabled; on the other, they selected Refinement 1 with session context disabled. In both cases, Feedback Method 2 was used as described above. For each task, the corresponding three faulty student submissions were used on both pages to ensure comparability over time.

The same set of evaluation questions was used in this second part, despite its different focus. This decision was made to maintain consistency across the evaluation and to enable direct comparison of results. While more tailored questions were considered, they risked becoming overly suggestive and potentially biasing participants' responses. Instead, the reuse of established questions allows shifts in response patterns to implicitly reveal the effects of Rubber Duck Debugging and session context, without introducing new variables through reworded questions.

Survey In the third part of the evaluation, participants' experience in Python, teaching, and additional programming languages was surveyed, as shown in Figure 4.3.

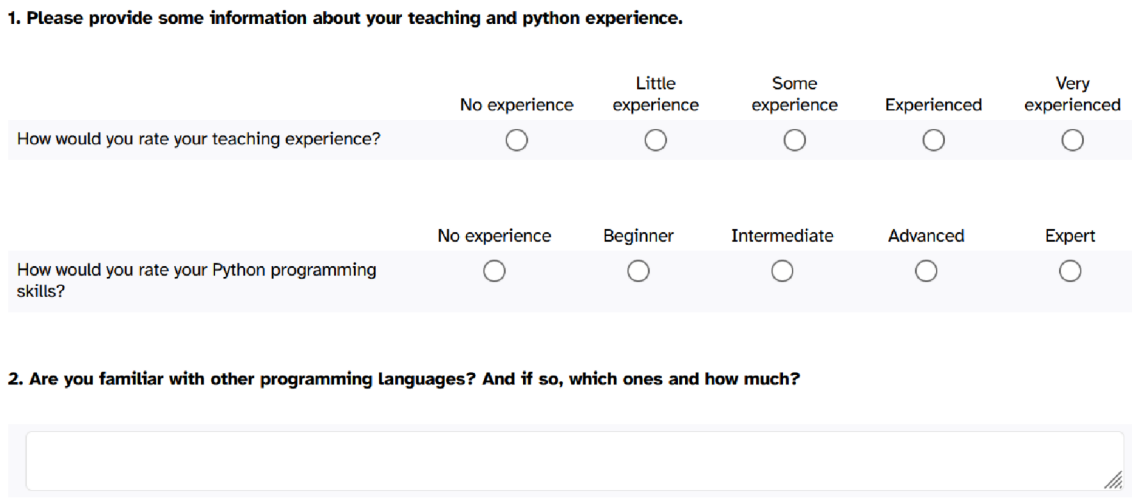


Figure 4.3: Survey Questions

4.2 Evaluating the Results

4.2.1 Quantitative Results

All participants reported prior Python and teaching experience. Python skill ratings ranged from 3 to 5 (Intermediate to Expert), while teaching experience ranged from 3 to 4, with one exception rated 1 (No experience). Consequently, the feedback was evaluated from the perspective of relatively experienced users—an important factor when interpreting both the quantitative and qualitative results.

As previously stated, the evaluation was structured into two parts: a comparison of different feedback generation methods, and an analysis of the Rubber Duck refinement combined with session context, measured against an LLM baseline. This structure is maintained in this quantitative analysis.
For each part, one representative task is presented to illustrate the key trends. The remaining tasks, which follow similar patterns, are included in the appendix (see **??**) for reference. While overall trends are consistent, the selected tasks display the clearest differences between methods.
In the quantitative evaluation, the graphs depict participants' responses to the Likert-scale questions. The survey was completed by seven participants. Missing values are attributed to participants selecting the "don't know" option, which was explicitly included to prevent forced responses.

Feedback Methods The first part of the evaluation compares four feedback methods using student submissions. The task *Vehicle Class* is selected as a representative example, as differences between feedback variants are most clearly visible here. Other tasks (*Average Price, Car Dictionary*) exhibit similar patterns and the diagrams of their Likert analysis are included in Appendix D.

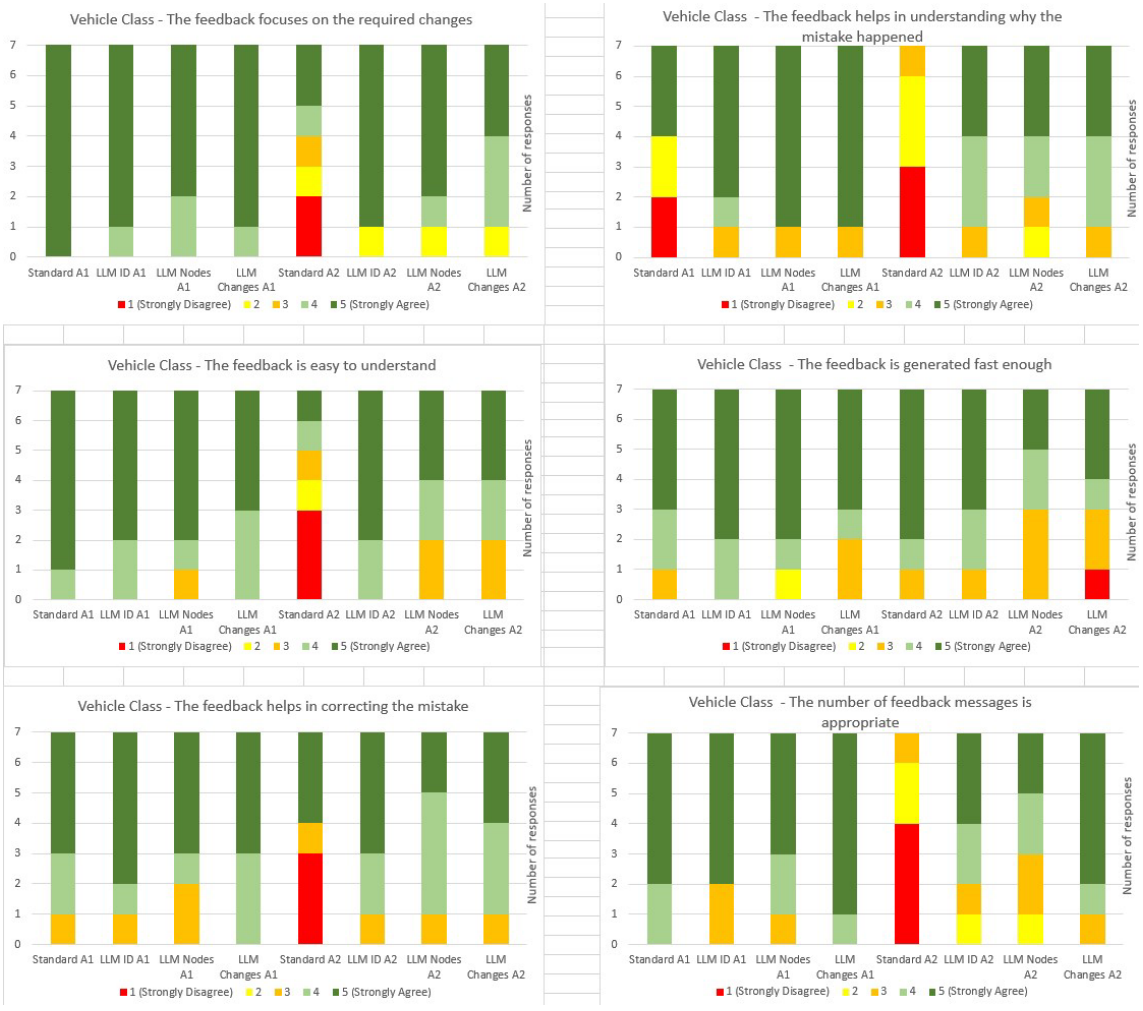


Figure 4.4: Evaluation Results for Task Vehicle Class

Figure 4.4 shows that across all six evaluation questions, the LLM-based feedback methods consistently receive higher ratings than the standard feedback method. This difference is particularly pronounced in submission 2, where the standard method scores lower across dimensions such as helpfulness, understandability, and clarity of the underlying mistake. Ratings for the standard method in this case frequently fall between 2 and 3, whereas LLM-based methods typically range between 4 and 5.
The LLM-based methods also demonstrate greater consistency between submissions 1 and 2. In contrast, the standard method varies more noticeably, both in overall ratings and in the distribution across response categories—particularly for questions related to focus and feedback quantity. In addition to higher average scores, most LLM variants show reduced variance in participant responses compared to the standard method. A minor exception is the *LLM Changes* variant, the third LLM-based method, which exhibits slightly more variability in responses to the speed-related question.
For most participants, feedback speed is rated positively across all methods. In the category *"The feedback is generated fast enough"*, most ratings lie above the midpoint of the scale, with only a few lower scores appearing for the *LLM Changes* variant.

Rubber Duck Refinement with Session Context The second part of the evaluation compares the LLM baseline feedback with the refined Rubber Duck approach, which incorporates session context. The task *ToDoList* is selected as the representative example, as it most clearly illustrates the differences between the two methods. As before, all six Likert-scale questions are considered. The results for the `Contact Book` task are included in Appendix D.
Across most questions, the LLM baseline achieves higher average scores than the Rubber Duck variant. This is particularly evident in the categories *"The feedback helps in correcting the mistake"* and *"The feedback helps me understand why the mistake happened"*, where the Rubber Duck method receives consistently lower ratings across all three submissions. Ratings for the baseline method tend to cluster around values 4 and 5, while the Rubber Duck method shows a broader spread, including several ratings of 2 and below.
Despite the use of sequential submissions and a persisted session context, the Rubber Duck condition showed no clear improvement across the three submissions. This indicates that the intended benefit of continuity and context between feedback rounds did not materialize within the scope of this evaluation.
With respect to perceived speed, both methods are rated similarly. While the first Rubber Duck submission receives a few lower ratings, subsequent submissions show no notable difference compared to the baseline. Likewise, the number of feedback messages is considered appropriate in both conditions, although the LLM baseline receives slightly higher scores overall.

4.2.2 Qualitative Results

Using the open-text fields, over 30 high-quality comments were collected. The most prominent themes are structured and presented in this section. Technical edge cases and minor ambiguities encountered during the evaluation are addressed separately in the limitations section.

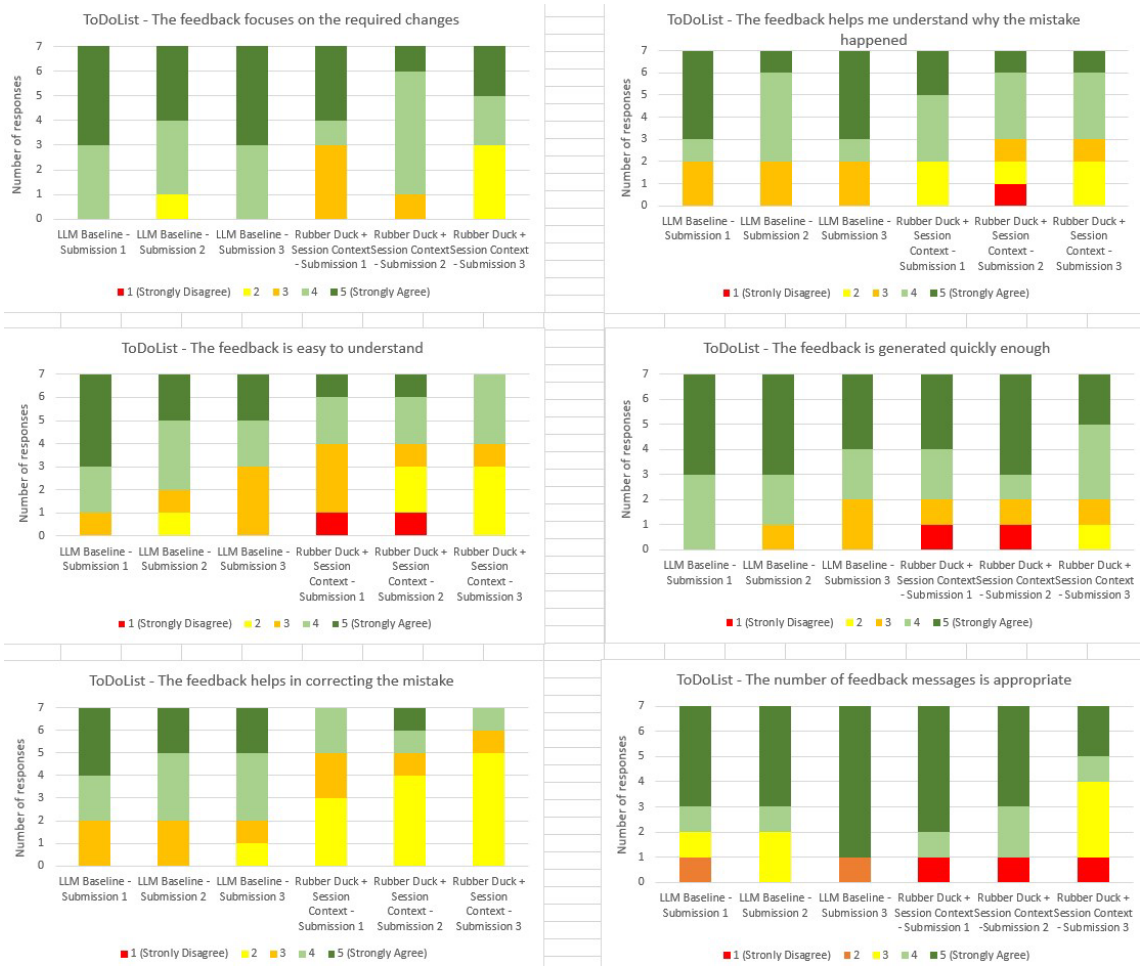


Figure 4.5: Evaluation Results for Task ToDoList

Line Hints The most frequent point of criticism concerned the line hints displayed beneath each feedback message. Several participants noted that these were often inaccurate—for example, stating that “line does not correspond to Error” [P5]. This issue was exclusively mentioned in relation to the LLM-based methods. In contrast, the line references for the standard feedback were generally accurate, although one participant remarked that the location indicators were overly verbose and harder to follow: “The location within the feedback is rather lengthy and harder to read” [P2].

Feedback Length Another commonly reported issue was the length of individual feedback messages. Comments such as “Too much text for my taste” [P1] suggest a preference for shorter, more concise responses. This concern was raised more frequently in the context of LLM-generated feedback. The standard method, by contrast, was rarely criticized for verbosity, but occasionally for the number of messages and some redundancy.

Correctness While the standard feedback was often seen as overly rigid or low-level, it was not explicitly described as incorrect. In contrast, LLM-generated feedback drew occasional criticism for factual inaccuracies or misleading suggestions. One participant noted, for instance, that in the first submission of the *Average Price* task,

the LLM incorrectly flagged the use of an additional function parameter, even though this was explicitly permitted by the task description.

Helpfulness Perceptions of helpfulness varied across methods. The standard feedback was sometimes described as more precise but less useful for understanding or resolving the underlying issue. For example, participants commented: “The feedback is not so helpful...” [P1] and “The first message is off-topic. overall [sic!] hard to understand” [P6]. Redundancy was also noted, particularly in longer tasks such as *Vehicle Class*, where P1 remarked: “I would not be able to correct anything with this feedback.” In contrast, LLM-based feedback was more often described as helpful for understanding the nature of the mistake. One participant observed: “The feedback helps me to understand the reasons behind the required changes but does not help in finding the lines to change. If it should help me learn to find the changes myself, it is appropriate” [P6].
Positive remarks also highlighted the LLM’s ability to include useful hints. For instance, one participant pointed out a case where the LLM provided the correct syntax for defining a set in Python ({}), rather than describing it abstractly.

Stylistic and Other Remarks A few comments addressed the tone and phrasing of the feedback. One participant noted that the messages addressed them too directly and should be formulated in a more neutral or general style.
In addition to the quantitative results, the Rubber Duck variant also received an individual remark, with P6 stating: “Giving feedback using questions seems odd.”
Overall, most of these observations apply broadly to all LLM-based methods. No consistent trends were identified that clearly distinguished the three LLM variants, apart from a few remarks indicating that the third variant (which includes code change context) had slightly longer response times—consistent with findings from the quantitative analysis.

4.2.3 Limitations

The evaluation yielded valuable insights into the effectiveness of different feedback methods. However, several limitations should be considered when interpreting the results.

Sample Size and Participant Profile The study involved seven participants, most of whom had at least intermediate programming experience. While this allowed for informed and critical evaluation of the feedback, the results may not generalize to absolute beginners—the intended target group of the feedback system. Future evaluations should include a broader spectrum of programming experience to better capture the needs of diverse user groups.

Study Design Constraints Participants did not interact with their own code but instead evaluated predefined submissions. This approach improved comparability between methods but may have reduced the personal relevance or engagement with the feedback—particularly in the case of rubber duck debugging, which relies on reflective engagement.

Moreover, the session context used in the second part of the evaluation was artificially constructed and limited in depth. While it introduced some continuity between submissions, it did not fully replicate the complexity of real-world feedback histories. A more extended and organically developed session history may have enabled more personalized or effective feedback.

Due to scope constraints, not all possible combinations of refinement techniques, feedback variants, and session context configurations were tested. It is possible that unexplored configurations would have yielded different or more nuanced results.

Time Pressure The structure of the evaluation led to more time being available for the earlier tasks compared to the later ones. This may have affected the level of engagement with more complex feedback types—such as rubber duck debugging—in the second part of the evaluation. Consequently, differences in perceived feedback quality may partly reflect variations in time and attention.

Rubber Duck-Specific Challenges The rubber duck debugging approach introduces a reflective, user-driven form of feedback. However, several participants found the feedback repetitive or of limited added value—likely because they were already familiar with the correct solutions due to the evaluation setup. Others expected more explicit guidance and found the indirect, question-based feedback style confusing.

Additionally, this method is inherently difficult to evaluate in a short-term context and may require repeated use to demonstrate its full value. These factors should be considered when interpreting the comparatively lower ratings for this condition.

Task and Feedback Inconsistencies A few isolated inconsistencies in task design and evaluation setup may have influenced participant perceptions. For instance, in one task, test cases expected a return value not clearly specified in the task description, potentially leading to misleading feedback. In another case, an indentation error in the provided code caused the feedback system to fail. While rare, such occurrences may have impacted the perceived reliability of certain feedback variants.

LLM-Specific Considerations A frequently reported issue in the qualitative feedback was the inaccuracy of line number references in LLM-generated messages. Several participants noted mismatches between the indicated line and the actual location of the error, which may have reduced the perceived utility of the feedback. In contrast, line hints from the standard method were generally accurate, though sometimes harder to interpret.

Finally, while all LLM outputs were generated using a fixed temperature of zero to minimize output variance, the generation process remains non-deterministic [OZHW23]. Minor variations across runs cannot be entirely ruled out and may have had a limited influence on the consistency and quality of the feedback.

4.2.4 Interpretation of Results

LLM Feedback Performs Consistently Better As expected, feedback generated by LLM-based methods received higher ratings than the standard method across nearly all submissions. These methods were perceived as more helpful and easier to understand, particularly in tasks such as *Vehicle Class*. Beyond average scores, LLM-based feedback also demonstrated greater consistency across different submissions, whereas the standard method exhibited higher variance. This supports the conclusion that LLMs not only provide more useful and comprehensible feedback overall but also do so more reliably across diverse task contexts. In direct response to the research question, LLM-based methods are clearly perceived as more useful and clearer than the standard method.

Correctness Remains a Limiting Factor Although LLM-generated feedback was generally well-received, participants occasionally identified misleading or incorrect suggestions. These issues were often subtle and not systematic but may still have reduced the perceived reliability of the feedback—especially for less experienced learners. In several cases, inaccuracies were linked to minor ambiguities in the task descriptions, particularly in the second part of the evaluation. These findings highlight that even when feedback is clear and well-formatted, doubts about correctness can undermine its perceived usefulness.

Rubber Duck Debugging Underperforms in the Given Setup The Rubber Duck refinement, combined with session context, consistently underperformed compared to the LLM baseline across all submissions in the second evaluation part. Participants frequently expressed confusion about the question-based format and found the reflective prompting unhelpful for small, familiar bugs. Since the evaluated submissions primarily contained minor errors, users expected more direct and actionable feedback. Under such conditions, reflective prompts appeared redundant. However, this does not preclude the potential of Rubber Duck feedback in more complex or uncertain problem-solving contexts. In the present implementation and task design, it was not perceived as useful or clear, and was rated less favorably than the baseline under these conditions.

Prompt Size Affects Model Performance Prompt violations occurred primarily with the Rubber Duck variant, suggesting that the combined use of session history and reflective prompting may have exceeded the effective input capacity of the `gpt-4o-mini` model. Other LLM variants, despite incorporating context, remained more coherent. This indicates that prompt size and structural complexity must be managed carefully. Exceeding the model's limits can negatively impact output clarity—an important consideration when designing feedback systems for educational purposes.

No Clear Differences Between LLM Variants—Except for Speed The three LLM-based feedback variants performed similarly in terms of perceived usefulness and clarity. However, the third variant, which included textual changes and required a secondary LLM call, was slower in generating responses. Despite the added complexity, it did not produce noticeably better feedback. This suggests that more elaborate

configurations do not necessarily improve output quality. In light of the research question, simpler LLM-based methods appear equally clear and useful, while offering greater efficiency in practical use.

Line Hints Show a Trade-off Between Brevity and Precision Participants noted that line hints in LLM-generated feedback were often imprecise or incorrect. This likely stems from the model being asked to generate line numbers based on high-level prompt information, without explicit structural anchoring in the code. Although the model had access to the full code (in parsed JSON form), outputting a single line number remains error-prone. By contrast, the standard method produced accurate line references, but these were embedded in verbose contextual strings that participants found harder to interpret.

A promising alternative would be to have the LLM generate short, targeted code snippets instead of numeric references. This could improve both clarity and usability by making the feedback easier to follow without overwhelming the user. This aligns with broader findings in LLM research suggesting that hallucinations are reduced when models are prompted to produce structured, contextual outputs rather than isolated factual statements—though prior work has primarily focused on verbal reasoning rather than numeric accuracy in code-related tasks [DKX+23].

Chapter 5 Future Work

The evaluation revealed several limitations and open questions that suggest opportunities for further development. This chapter outlines potential directions for addressing these challenges, with a particular focus on enhancing feedback quality, adapting to user needs, and increasing system flexibility.

Evaluating and Matching Large Language Models The current implementation relies on `gpt-4o-mini`, a lightweight model that provides solid baseline performance. However, its limitations became apparent in feedback scenarios involving complex prompts or extended context. Future work could investigate the use of more capable models—such as `gpt-4o` or `o3-mini`[1] —to assess whether they offer improved support for reasoning-intensive feedback tasks. Stronger reasoning capabilities may help close current quality gaps, particularly in complex or reflective scenarios such as rubber duck debugging, and in handling longer or structurally rich prompts. Depending on the nature of the feedback, whether it involves direct corrections or reflective guidance, different models may present better trade-offs between latency and quality. A more dynamic approach could assign models based on the feedback type or task complexity.

Context-Sensitive Use of Reflective Feedback Although reflective feedback techniques like rubber duck debugging underperformed in the current evaluation setting, their potential remains promising in alternative contexts. Such methods may be more effective when learners experience persistent difficulties or uncertainty in their approach. In these cases, prompting users to articulate their reasoning or reflect on their code could be beneficial—especially when supported by a more capable model. A system that activates this feedback mode only under certain conditions could help minimize cognitive overhead while providing meaningful support when most needed.

Session Context and User Continuity Meaningful session-based feedback depends on persistent user context over time. In the current setup, submissions were processed with access to previous context, but this context was only retained for a limited duration—specifically, for a few hours—without any persistent user identification. To enable more advanced context-aware strategies, future versions of the system could incorporate lightweight user management and support longer-term session continuity

[1] `o3-mini` and `gpt-4o` are other models provided by OpenAI. See `https://platform.openai.com/docs/models` for details.

across exercises. This would allow feedback to evolve with the learner's progress and provide more personalized guidance over a longer period of time.

Proactive and Adaptive Feedback Strategies Currently, feedback is only generated when explicitly requested. Introducing mechanisms for proactive feedback—triggered by factors such as inactivity, repeated errors, or the magnitude of recent changes—could help learners identify issues earlier. Determining the appropriate feedback timing in such scenarios will be crucial to avoid distraction or cognitive overload. Additionally, the system could dynamically select appropriate feedback methods or refinements based on task characteristics or observed user behavior. Feedback verbosity might also be adapted to the learner's self-assessed experience level, ensuring alignment with user expectations.

Improving Line-Level Precision Line number references in LLM-generated feedback were not always reliable, despite the availability of structured code input. This limitation could be addressed by replacing numeric line indicators with short code excerpts that point learners to relevant locations. The standard method already implements this using highly detailed, line-specific snippets—which are precise but often overly verbose. In contrast, LLM-based feedback tends to be more concise but lacks precision. A middle ground—concise, targeted excerpts grounded in the actual code structure—could combine the strengths of both approaches, improving clarity while minimizing hallucinated or incorrect references. This may also be more intuitive for beginners to interpret.

Refining Reference Solution Selection The current approach selects and persists a reference solution upon the first submission, regardless of how well it matches the learner's strategy. This can lead to suboptimal feedback if the chosen solution diverges significantly from the learner's path. A more robust approach would condition the persistence of the initial reference solution on its relative closeness—only storing it if its distance to the next-best reference falls below a defined threshold. Otherwise, the system could delay selection until a clearer match emerges. This would prevent early misalignment and help ensure that feedback remains relevant throughout the session.

Supporting Syntax Error Feedback Although the system currently focuses on semantic errors, syntax error feedback could be integrated with minimal overhead. LLMs are capable of generating more approachable and explanatory messages than standard interpreter errors and may even connect syntax mistakes to common beginner misconceptions. Incorporating this type of feedback would help provide a more complete and supportive learning experience, particularly for users in the early stages of programming.

Chapter 6 Conclusion

This thesis extended the existing Pyrat system by integrating additional feedback generation methods and a session context mechanism aimed at enabling more adaptive and context-aware feedback. The session context was designed to persist both user submissions and corresponding feedback, with the goal of providing more coherent responses over time. Three LLM-based feedback methods were implemented, varying in the type and extent of context provided to the model: one based solely on the closest reference solution, one additionally incorporating structural information about the user's code (i.e., buggy code nodes), and one also using the final textual changes that the standard method would generate. In addition, two refinements were explored: a selection-based refinement that limited the feedback scope, and a reflective refinement based on the rubber duck debugging technique.

The evaluation aimed to compare these methods in terms of perceived usefulness and clarity, as posed in the research question: *How do different feedback generation methods in Pyrat compare in terms of perceived usefulness and clarity?* In direct response to this question, the evaluation shows that LLM-based feedback methods were consistently rated as more useful and clearer than the original, rule-based approach. They were also perceived as more consistent between submissions, likely due to the LLMs' ability to generalize across similar tasks. Notably, the lightweight model used in this thesis—`gpt-4o-mini`—achieved strong results, suggesting that targeted context combined with well-crafted prompts can enable high-quality feedback even under constrained conditions. No substantial differences in perceived usefulness or clarity were observed between the three LLM variants, despite differences in contextual input. The most complex variant, which included textual changes, led to slower response times but did not yield significantly better feedback. The rubber duck refinement underperformed in this setting. Although its reflective nature holds theoretical promise, it was not well received by participants—likely due to a mismatch between its format and the types of errors or expectations involved.

These findings suggest that LLM-based feedback is not only feasible within Pyrat, but offers clear benefits for user experience. The fact that strong results were achieved with a general-purpose, lightweight model further supports the practicality of LLM integration in educational tools without incurring prohibitive costs. While correctness remains a known limitation of LLMs, inaccuracies in this evaluation were present but not widespread and did not appear to negatively impact the perceived quality of the feedback. For reflective strategies like rubber duck debugging, the timing and context of their application seem to be critical. Such strategies may be more effective in specific learning contexts—for example, when learners are stuck or unsure how

to proceed. Alternatively, a reversed approach may also be valuable: using less revealing, reflective prompts like rubber duck debugging for early feedback requests to foster independent reasoning, followed by more explicit guidance as needed.[1]

This study has limitations, including a small sample size, short evaluation sessions, and a participant group that does not fully represent the intended beginner audience. Furthermore, the session context used during evaluation was artificially constructed and did not reflect a natural interaction workflow.

Looking ahead, future development should focus on strengthening the existing implementation—e.g., in terms of robustness and performance, testing more capable LLMs, and adapting feedback strategies to different learning contexts. With this foundation established, more advanced features—such as proactive feedback mechanisms—can be explored.

[1] See e.g., Xiao et al. [XHS24] for staged feedback strategies in LLM-based tutoring.

Appendix A Bibliography

[BBB+19] Douglas S Blank, David Bourgin, Alexander Brown, Matthias Bussonnier, Jonathan Frederic, Brian Granger, Thomas L Griffiths, Jessica Hamrick, Kyle Kelley, M Pacer, et al. nbgrader: A tool for creating and grading assignments in the jupyter notebook. *The Journal of Open Source Education*, 2(11), 2019.

[BLC14] Hyunjung Byun, Jung Lee, and Frank A Cerreto. Relative effects of three questioning strategies in ill-structured, small group problem solving. *Instructional Science*, 42:229–250, 2014.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[CBL+89] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science*, 13(2):145–182, 1989.

[CLS+25] Zhongpu Chen, Yinfeng Liu, Long Shi, Zhi-Jie Wang, Xingyan Chen, Yu Zhao, and Fuji Ren. Mdeval: Evaluating and enhancing markdown awareness in large language models, 2025.

[DKX+23] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models, 2023.

[Esp24] Francesco Esposito. *Programming Large Language Models with Azure Open AI: Conversational Programming and Prompt Engineering with LLMs*. Microsoft Press, 2024.

[GRZ18] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4):465–480, 2018.

[HAM+19] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 388–398. IEEE, 2019.

[HL24] Hasnain Heickal and Andrew Lan. Generating feedback-ladders for logical errors in programming using large language models. *arXiv preprint arXiv:2405.00302*, 2024.

[Hun00] Andrew Hunt. *The pragmatic programmer*. Pearson Education India, 1900.

[Jup24] Project Jupyter. Project jupyter, June 2024. Last accessed: 2025-12-03.

[Kin91] Alison King. Effects of training in strategic questioning on children's problem-solving performance. *Journal of Educational psychology*, 83(3):307, 1991.

[KN16] Angelo Kyrilov and David C Noelle. Do students need detailed feedback on programming exercises and can automated assessment systems provide it? *Journal of Computing Sciences in Colleges*, 31(4):115–121, 2016.

[KYW+24] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In *Proceedings of the 2024 chi conference on human factors in computing systems*, pages 1–20, 2024.

[LKK25] Dominic Lohr, Hieke Keuning, and Natalie Kiesler. You're (not) my type-can llms generate feedback of specific types for introductory programming tasks? *Journal of Computer Assisted Learning*, 41(1):e13107, 2025.

[OBCR23] Priti Oli, Rabin Banjade, Jeevan Chapagain, and Vasile Rus. Automated assessment of students' code comprehension using llms. *arXiv preprint arXiv:2401.05399*, 2023.

[OZHW23] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. Llm is like a box of chocolates: the non-determinism of chatgpt in code generation. *arXiv e-prints*, pages arXiv–2308, 2023.

[PA16] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[Ste24] Leonard Steffes. Contextual automatic code repair for Python programming novices. Masterarbeit, RWTH Aachen University, Aachen, 2024. Veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Masterarbeit, RWTH Aachen University, 2024.

[TWH+24] Xiaohang Tang, Sam Wong, Marcus Huynh, Zicheng He, Yalong Yang, and Yan Chen. Sphere: Scaling personalized feedback in programming classrooms with structured review of llm outputs. *arXiv preprint arXiv:2410.16513*, 2024.

[WMP24] Juliette Woodrow, Ali Malik, and Chris Piech. Ai teaches the art of elegant coding: Timely, fair, and helpful style feedback in a global course. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 1442–1448. ACM, March 2024.

[WWS+22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[XHS24] Ruiwei Xiao, Xinying Hou, and John Stamper. Exploring how multiple levels of gpt-generated programming hints support or disappoint novices. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI '24, page 1–10. ACM, May 2024.

[ZMH+23] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers, 2023.

Appendix B Evaluation Codes

The codes for the tasks Average Price, Car Dictionary, and Vehicle Class can be found in the appendix B of the master's thesis on which this thesis builds [Ste24].
The following buggy submissions for the `To-Do List` and `Contact Book` tasks were used in the evaluation.

```python
class TodoList:
    def __init__(self):
        self.tasks = []

    def add_task(self, description):
        self.tasks.append({'description': description, 'completed': False})
        return f"Added task: {description}"

    def list_tasks(self):
        output = []
        for i, task in enumerate(self.tasks, 1):
            status = "Done" if task['completed'] else "Pending"
            output.append(f"{i}. {task['description']} - {status}")
        return "\n".join(output)

    def remove_task(self, index):
        if self.tasks:
            removed = self.tasks.pop(0)
            return f"Removed task: {removed['description']}"
        else:
            return "No tasks to remove."
```

Listing B.1: To-Do List 1

```python
class TodoList:
    def __init__(self):
        self.tasks = []

    def add_task(self, description):
        self.tasks.append({'description': description, 'completed': False})

    def list_tasks(self):
        for i, task in enumerate(self.tasks):
            status = "Done" if task['completed'] else "Pending"
            print(f"{i+1}. {task['description']} - {status}")

    def remove_task(self, index):
        if index <= len(self.tasks):
            removed = self.tasks.pop(index)
            print(f"Removed task: {removed['description']}")
        else:
            print("Invalid task index.")
```

Listing B.2: To-Do List 2

```python
class TodoList:
    def __init__(self):
        self.tasks = []

    def add_task(self, description):
        self.tasks.append({'description': description, 'completed': False})
        return f"Added task: {description}"

    def list_tasks(self):
        output = []
        for i, task in enumerate(self.tasks, 1):
            status = "Done" if task['completed'] else "Pending"
            output.append(f"{i}. {task['description']} - {status}")
        return "\n".join(output)

    def remove_task(self, index):
        found = False
        for i in range(len(self.tasks)):
            if i + 1 == index:
                removed = self.tasks.pop(i)
                found = True
                messages.append(f"Removed task: {removed['description']}")
            else:
                messages.append("Invalid task index.")
        if not found:
            messages.append("Task index not found.")
        return "\n".join(messages)
```

Listing B.3: To-Do List 3

```python
class ContactBook:
    def __init__(self, contacts=None):
        self.contacts = contacts if contacts is not None else []

    def add_contact(self, name, phone):
        self.contacts.append({'name': name, 'phone': phone})
        return f"Added contact: {name}"

    def find_contact(self, name):
        for contact in self.contacts:
            if contact['name'] == name:
                return contact
        return None

    def remove_contact(self, name):
        for contact in self.contacts:
            if contact['name'] == name:
                self.contacts.remove(contact)
        return "Contact not found"
```

Listing B.4: Contact Book 1

```python
class ContactBook:
    def __init__(self, contacts=None):
        self.contacts = contacts if contacts is not None else []

    def add_contact(self, name, phone):
        self.contacts.append({'name': name, 'phone': phone})
        return f"Added contact: {name}"

    def find_contact(self, name):
        for contact in self.contacts:
            if contact['name'] == name:
                return contact
        return None

    def remove_contact(self, name):
        for contact in self.contacts:
            if contact['name'] == name:
                self.contacts.remove(contact)
                return f"Removed contact {name}"
            else:
                return "Contact not found"
```

Listing B.5: Contact Book 2

```python
class ContactBook:
    def __init__(self, contacts=None):
        self.contacts = contacts if contacts is not None else []

    def add_contact(self, name, phone):
        self.contacts.append({'name': name, 'phone': phone})
        return f"Added contact: {name}"

    def find_contact(self, name):
        for contact in self.contacts:
            if contact['name'] == name:
                return contact
        return None

    def remove_contact(self, name):
        removed = False
        messages = []
        for contact in self.contacts:
            if contact['name'] == name:
                self.contacts.remove(contact)
                removed = True
                messages.append(f"Removed contact {name}")
                break
            else:
                messages.append("Contact not found")
        if not removed:
            messages.append("Contact not found")
        return "\n".join(messages)
```

Listing B.6: Contact Book 3

Appendix C Swagger Documentation of `/api/check/session` Endpoint

```
/**
 * @swagger
 * /api/check/session/{task_id}:
 *  post:
 *   summary: Evaluate code submission for a specific task with session context
 *   parameters:
 *    - in: path
 *      name: task_id
 *      required: true
 *      schema:
 *       type: string
 *       description: The ID of the task to evaluate
 *    - in: cookie
 *      name: sessionId
 *      required: false
 *      schema:
 *       type: string
 *       description: Optional session ID for maintaining user context
 *    - in: query
 *      name: feedbackMethod
 *      required: false
 *      schema:
 *       type: string
 *       enum: [standard, llmGenerateOnId, llmGenerateOnIdNodes,
    llmGenerateOnIdNodesChanges]
 *       default: standard
 *       description: Feedback generation method
 *    - in: query
 *      name: feedbackRefinement
 *      required: false
 *      schema:
 *       type: string
 *       enum: [select, rubberDuck]
 *       description: Optional refinement of feedback (LLM-selected messages or
     Rubber Duck Debugging)
 *   requestBody:
```

```
*     required: true
*     content:
*       application/json:
*         schema:
*           type: object
*           properties:
*             code:
*               type: string
*               description: The code to be evaluated
*   responses:
*    200:
*     description: Feedback generation successful
*     headers:
*      Set-Cookie:
*       schema:
*        type: string
*        description: Session ID cookie that is returned or updated
*     content:
*      application/json:
*       schema:
*        type: object
*        properties:
*         result:
*          type: object
*          description: The evaluation result
*         sessionId:
*          type: string
*          description: The session ID returned
*    401:
*     description: Unauthorized (invalid or expired session ID)
*    500:
*     description: Internal server error
*/
```

Listing C.1: Swagger Documentation for Session Endpoint

Appendix D Further Evaluation Results

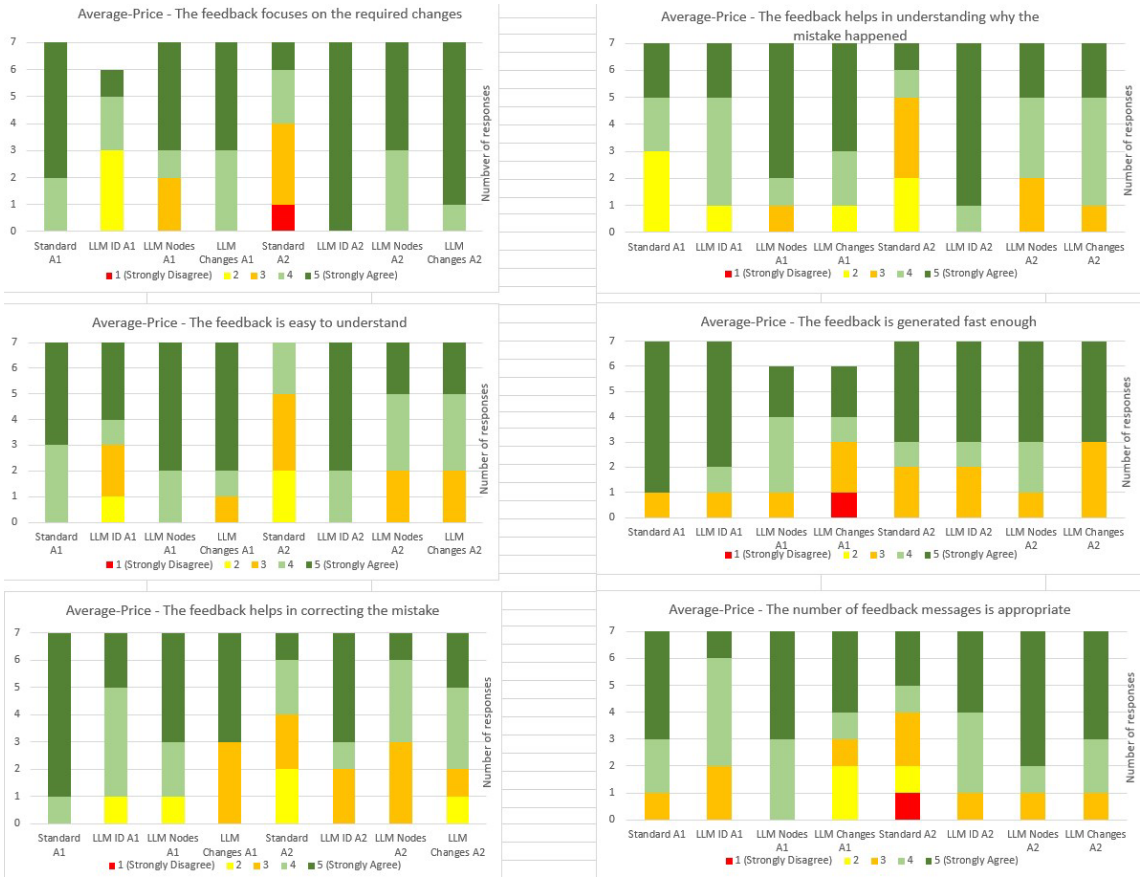


Figure D.1: Evaluation Results for Task Average Price

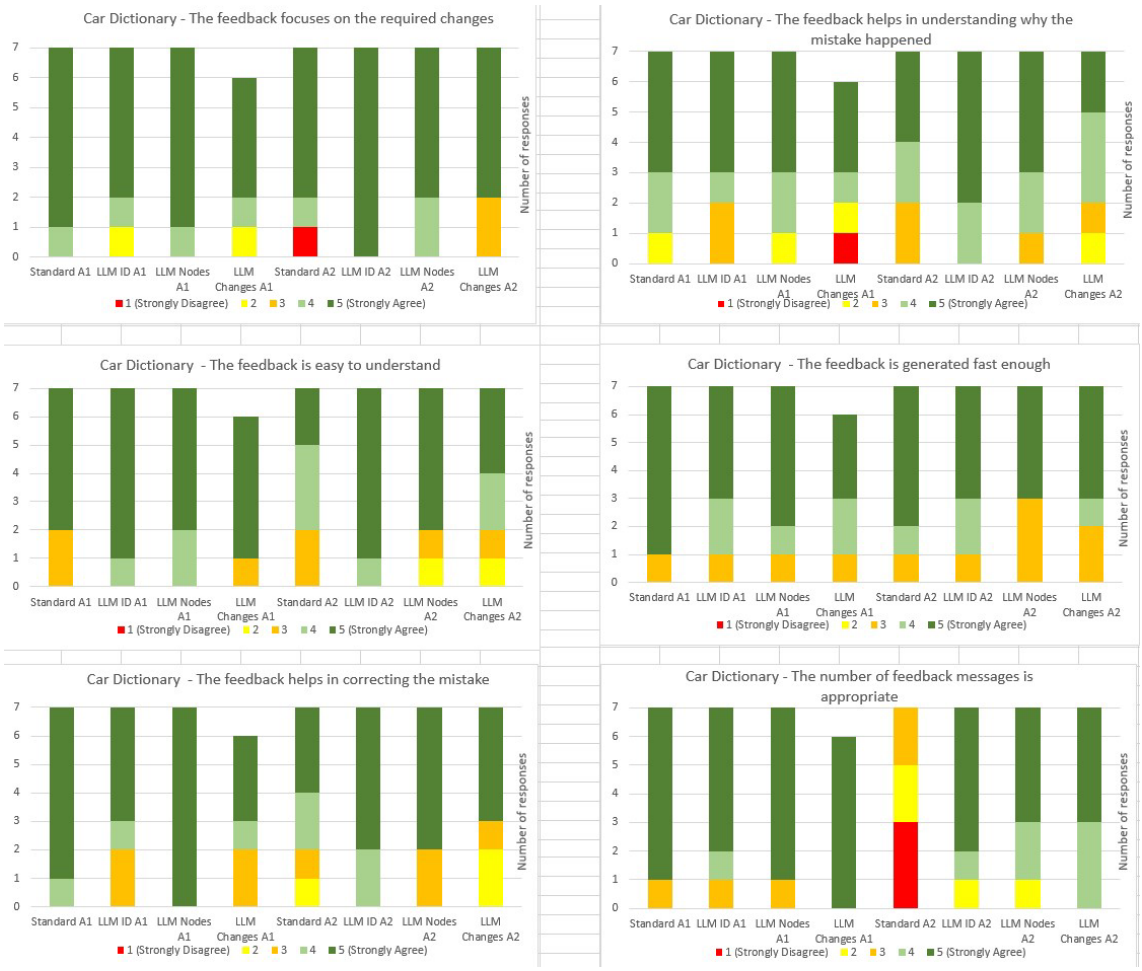


Figure D.2: Evaluation Results for Task Car Dictionary



Figure D.3: Evaluation Results for Task Contact Book

Appendix E Digital Appendix

The digital appendix is submitted on an SD card. To get access to this material they shall contact the author or LuFG i9.

Eidesstattliche Versicherung
Declaration of Academic Integrity

Herr, Yuval
Name, Vorname/Last Name, First Name

433722
Matrikelnummer (freiwillige Angabe)
Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/Masterarbeit* mit dem Titel
I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis* entitled

Extension and Evaluation of Feedback Mechanisms in the Python Repair Tool Pyrat through the Use of LLMs

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt; dies umfasst insbesondere auch Software und Dienste zur Sprach-, Text- und Medienproduktion. Ich erkläre, dass für den Fall, dass die Arbeit in unterschiedlichen Formen eingereicht wird (z.B. elektronisch, gedruckt, geplottet, auf einem Datenträger) alle eingereichten Versionen vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without unauthorized assistance from third parties (in particular academic ghostwriting. I have not used any other sources or aids than those indicated; this includes in particular software and services for language, text, and media production. In the event that the work is submitted in different formats (e.g. electronically, printed, plotted, on a data carrier), I declare that all the submitted versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

Aachen, September 4, 2025
Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen/Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
§ 156 StGB (German Criminal Code): False Unsworn Declarations
Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
§ 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence
(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding one year or to a fine.
(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, September 4, 2025
Ort, Datum/City, Date

Unterschrift/Signature