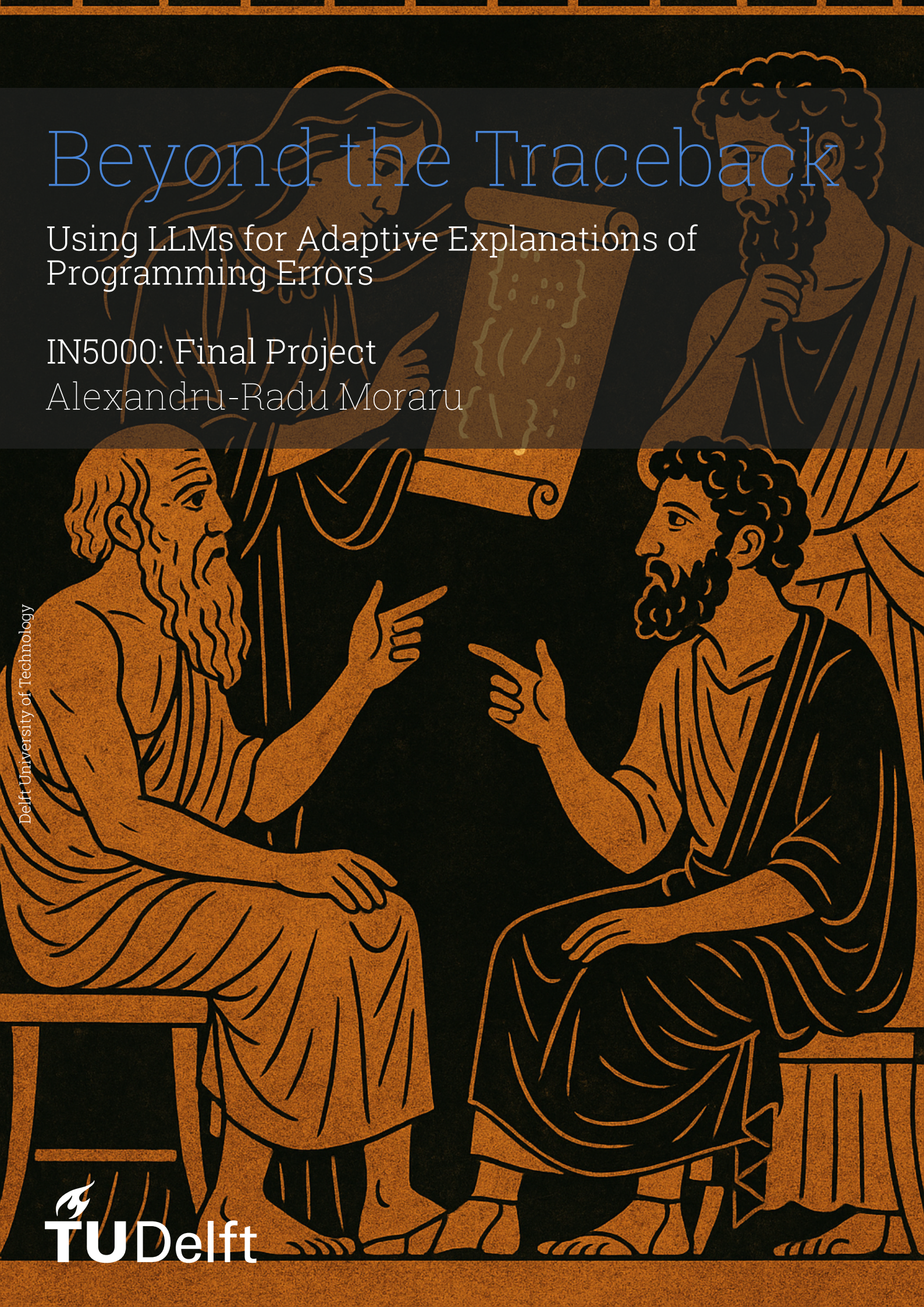
Beyond the Traceback

Using LLMs for Adaptive Explanations of Programming Errors

IN5000: Final Project

Alexandru-Radu Moraru



Delft University of Technology

TU Delft

Beyond the Traceback

Using LLMs for Adaptive Explanations of Programming Errors

by

Alexandru-Radu Moraru

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended on Friday August 29, 2025 at 1:00 PM CEST

Student number:	4798961	
Research Group:	Web Information Systems	
Project duration:	January 6, 2025 – August 29, 2025	
Thesis committee:	ir. S. Biswas,	TU Delft, Daily Co-Supervisor
	Dr. ir. U. Gadiraju,	TU Delft, Thesis Advisor
	Dr. ir. P. Pawelczak	TU Delft, External Committee Member

Cover: Illustration of debating philosophers in the style of ancient Greek vase artwork, generated with ChatGPT

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

In line with existing claims in literature about the current state of programming error messages, I too often find them frustrating and insufficiently clear, having been "blessed" by genuinely intricate bugs and lack of error message specificity both during my time as a student, and during my professional career. This, coupled with my reluctance at first, later replaced by an ever-increasing enthusiasm for LLMs, has led me to pursue this project, where I investigate the effectiveness of skill-adaptive LLM-enhanced error messages. I hope that these findings, whatever their size, will have a positive impact on error message design and will continue to inspire further research on this topic.

Acknowledgments

Over the past three years, pursuing an MSc degree while workly mostly full-time has been one of the most (if not the most) challenging task of my academic and professional career. Although I am not certain I would choose the same path again, I am proud of having seen it through and of the immense discipline I discovered in myself. Still, the credit is not mine alone. Without the support of many people, I would not be writing this preface today.

Back in 2021, I took the Human-Computer Interaction elective in my Computer Science and Engineering bachelor's at TU Delft, and had the pleasure of learning from Ujwal Gadiraju. I remember thinking that if I ever pursued an MSc at TU Delft, I would be grateful to have him as my thesis advisor. After graduating in 2021 amid the pandemic, the timing didn't feel right for further study, so I started working in industry as a software engineer. Fast-forward three years to 2024, I reached out to Ujwal, we shared interest in this topic, and the collaboration began. Thank you for the very generous guidance and advice; I learned a great deal from you.

I am equally grateful to Shreyan Biswas, who supervised me over the past nine months. Our weekly, and sometimes twice a week sessions were crucial in shaping this work. Thank you for helping me sift through the noise and for pushing me to investigate key aspects that might otherwise have been left underexplored. Finally, I am also extending my appreciation to P. Pawelczak, for serving on my thesis commitee.

In addition, I'm especially thankful to my colleagues at TomTom for the trust, flexibility, and encouragement that made it feasible to balance work with my studies. To that end, I'm sending a big thank you to Loredan, Andy, Patrick, Hristo, Monika, Andreas, Michael, Andrzej and Simon. These past four years have taught me a lot, thanks to all of you. I'm particularly thankful for our online/offline discussions on so many diverse and interesting subjects. On a more personal note, to my friends outside work–Sadry, Davia, Mark, Andrei, Mara, Alex–thank you for sticking by me all these years (*some of you more than a decade*!), even when I reluctantly had to cancel plans to dedicate more time to exams or this thesis.

That being said, I would have never been able to achieve even half of the things I did, were it not for Ana, who I blessedly, and arguably very randomly, met in Groningen almost 5 years ago. Thank you for being part of my life. Your love, kindness, and colossal amount of patience have been a constant source of light and motivation all of these years. Last but not least, to my parents and grandparents, both those with us and who we *dearly miss*, none of this would have been possible without you..."*multumesc*!". To my dad, thank you for teaching me how to be pragmatic and calm, even in the face of adversity. And to my mom, thank you for showing me how to view life in a colorful, yet whimsical way.

Alexandru-Radu Moraru
Delft, August 2025

Summary

Error messages are a primary feedback channel in programming environments, yet they often obstruct progress, especially for novices. Although large language models (LLMs) are widely used for code generation and debugging assistance, there is limited empirical evidence that LLM-rephrased error messages consistently improve code correction capabilities, and skill-adaptive designs remain largely unexplored. We introduce a framework that uses an LLM to rewrite Python standard interpreter errors in two different styles, which are designed to be tailored to user expertise: the pragmatic style, which is concise and action oriented, and the contingent style, which provides scaffolded, actionable guidance organized by a clear argumentation model. To measure Python skill level reliably, we first ran a pilot study that informed the design of a short 8 multiple-choice question assessment focused on debugging and error-message interpretation. We then used this instrument in the main study to classify participants as either novices or experts.

To gauge the effectiveness of our LLM-enhanced programming error messages (PEMs), we evaluated the framework in a crowdsourced Prolific study with 103 participants. We measured objective outcomes such as fix rate, time to fix, and number of attempts to fix, while also capturing subjective perceptions of PEMs, including readability, cognitive load, and authoritativeness. Objectively, LLM-enhanced PEMs showed favorable trends but did not produce statistically significant improvements over the standard interpreter. Subjectively, novices and experts alike, rated the pragmatic messages as significantly more readable and helpful, lower in intrinsic and extraneous cognitive load, and considerably less authoritative. Contingent messages exceeded the baseline on average but did not consistently reach statistical significance across all of our measurements, which points to a need for tighter control of error message verbosity and granularity, particularly for beginners.

These results show that LLMs, especially small-sized ones, are already capable of delivering targeted text-rewriting interventions that improve the perceived quality of error feedback. Future work should validate the effects at larger scale and across languages, expand coverage of real-world error contexts, and pursue true adaptivity in which error message style and level of detail adjust dynamically to user skill and task state.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

Abbreviation	Definition
ACID	Atomicity, Consistency, Isolation, and Durability
AI	Artificial Intelligence
API	Application Programming Interface
CER	Claim, Evidence, and Reasoning
CLT	Cognitive Load Theory
CI/CD	Continuous Integration and Continuous Delivery
CS1/CS2	Computer Science 1/2
CTT	Classical Test Theory
DevEx	Developer Experience
GenAI	Generative Artificial Intelligence
GHA	GitHub Actions
GPA	Grade Point Average
GPT	Generative Pre-trained Transformer
HCI	Human-Computer Interaction
IDE	Integrated Development Environment
IRT	Item Response Theory
I/O	Input/Output
LLM	Large Language Model
MCQ	Multiple-Choice Question
NLP	Natural Language Processing
OOP	Object-Oriented Programming
PEM	Programming Error Message
RAG	Retrieval Augmented Generation
RCT	Randomized Controlled Trial
RQ	Research Question
SaaS	Software as a Service
SD	Standard Deviation
SoTA	State of The Art
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine
YoE	Years of Experience

1 Introduction

Learning from mistakes is a fundamental aspect of human cognition and development. This principle applies not only to individuals, but also to organizations and societies. Throughout life, challenges and obstacles force us to adapt, refine our approaches, and learn from experience. In programming, learning through mistakes is remarkably common, from novice programmers dealing with simple syntax errors to seasoned developers facing complex architectural flaws in large-scale software systems. The ability to learn from errors is essential for both personal growth and professional advancement in software development.

Although the process can be frustrating, mistakes are a powerful driver of improvement and innovation. However, not all mistakes carry the same weight. Typographical errors or minor syntax mistakes may be easily forgiven, particularly when made by beginners who are still learning the basics, while other errors can have far-reaching consequences [50, 24], potentially leading to fatal outcomes [20]. Regardless of their severity, mistakes, especially compiler and interpreter errors, are an unavoidable part of programming. Even with the emergence of large language models (LLMs) capable of generating syntactically-correct code, building SaaS applications from natural language prompts, and assisting with debugging, these systems remain fallible. Ultimately, the responsibility for code correctness rests with the programmer, who must still engage in the often tedious process of identifying and fixing errors.

Despite decades of progress in programming language design, error messages remain notoriously difficult to interpret. Novice programmers often struggle to understand both their meaning and how to act on them, while experienced developers, though better equipped, still invest significant effort in parsing and applying the information they contain. Several studies [1, 46, 48] confirm that debugging remains among the most time-consuming, challenging, and irritating aspects of software development, even in professional environments. Extended debugging times slow feature delivery, inflate costs, reduce productivity, and harm developer morale [67]. In industry settings, this can lead to missed deadlines, budget overruns, and delayed releases [73]. In educational contexts, this can harm learners' confidence and ultimately lead to higher attrition rates in programming courses [70, 27].

A notable literature survey by Becker et al. [9] identified several persistent challenges in error message design, including excessive technical jargon, insufficient contextual information, and a lack of clear and actionable guidance. The authors emphasize the need for error messages that are more accessible to the reader, supportive of learning, and less likely to hinder problem solving. Although various approaches have been proposed to address some of these concerns in isolation [68, 40, 64], no unified framework exists yet that is both theoretically grounded and practically applicable to creating user-adaptive, LLM-enhanced programming error messages (PEMs).

This thesis report proposes such a framework, grounded in the theory of actionable insights, which aims to rephrase error messages in ways that are (1) pragmatically useful, (2) contingent to both program structure and the user's expertise level, (3) reader-friendly, (4) less cognitively demanding and more empathetic. In this context, we define pragmatic error messages as those that directly support problem resolution, while contingent messages are those that provide more contextual information. We argue

that such an approach can reduce the debugging overhead and enhance the learning experience, ultimately enabling more efficient and effective programming practices.

By leveraging LLMs as adaptive tutors, this research investigates how error messages can be dynamically rephrased and formatted to meet the needs of both novice and expert programmers. We conduct a crowdsourced evaluation of the proposed framework, assessing whether contingent and pragmatic error messages generated by LLMs can address the pain points identified by Becker et al. [9], namely poor readability, excessive cognitive load, lack of context, limited actionability, and an intimidating tone.

1.1. Contributions

The main contributions of our research are as follows:

- We conducted an empirical investigation of Python error messages and debugging, then developed through a pilot study a multiple-choice Python skill assessment, focused on debugging and error messages, which was used as a more reliable proxy than self-reported proficiency.
- A skill-contingent framework for rephrasing compiler and interpreter error messages, created for both novice and expert programmers. Although designed for Python, the framework is readily adaptable to other programming languages, with some modifications required before validation.
- We conducted a crowd-sourced evaluation on Prolific measuring the impact of our framework on bug fixing performance, as well as perceived readability, cognitive load, and authoritativeness.
- Open-source release of all research artifacts, including code, datasets, evaluation pipelines, the deployed web application used for the Prolific study, and a repository that can be used for benchmarking error message styles across diverse code snippets [45].

1.2. Research Questions

Error messages are among the most frequent feedback channels in programming, yet they often hinder progress, especially for beginners. Despite rapid advancements in applying LLMs to software engineering, there is little empirical evidence that LLM-generated error messages improve objective code-correction, and skill-adaptive designs remain largely untested. To the best of our knowledge, this is among the first controlled studies to evaluate skill-adaptive LLM-rephrased PEMs, jointly examining objective and subjective outcomes. This motivates us to formulate the following research questions:

- ***RQ1*: How do LLM-powered skill-adaptive Python error messages influence novice and expert programmers' code-correction performance in comparison to standard interpreter messages?**
  Beyond learning, error messages' primary goal is to guide code repair. By tracking fix rate (i.e., proportion of snippets corrected per error message style), time-to-fix, or number of attempts to fix, we assess whether LLMs cut debugging overhead and improve code comprehension.
- ***RQ2*: How do novice and expert programmers perceive the readability, cognitive load, and authoritativeness of skill-adaptive, LLM-powered Python error messages relative to the standard interpreter messages?**
  While the primary aim of error messages is code repair, we also examine programmers' perceptions, namely helpfulness, readability, and clarity, especially for novices who may struggle with the jargon and complexity of standard messages. These subjective ratings complement objective outcomes, yielding a more holistic view of our styles against the standard interpreter output.

To answer these research questions, we considered two LLM-powered message styles, pragmatic and contingent, initially inspired by a student-led pilot study, which was launched as part of the Crowd Computing MSc course at TU Delft. When taking a close look at our experimental results, while mean correction rates favored our skill-adaptive messages, differences were ultimately not statistically significant. By contrast, pragmatic messages yielded significantly lower cognitive load and higher perceived helpfulness and readability, while being seen as less authoritative for novices and experts alike. Contingent PEMs trended better on most metrics but were, to our surprise, mostly non-significant. Although LLM choice and performance lay outside of our RQs, the fact that these effects were achieved with an 8B parameter model highlights the feasibility of small models for targeted, text-rewriting interventions.

1.3. Thesis Outline

We begin in Chapter 2 with a review of relevant background literature and related work, highlighting connections to our study where appropriate. Chapter 3 then outlines the research methodology, covering the Python skill-assessment pilot study, the second pilot for selecting candidate code snippets and error messages for the main study, the design of the PEM rephrasing framework, and the development of the Prolific web application for crowdsourced evaluation. The experimental setup of the main study is formalized in Chapter 4, including independent and dependent variables, potential confounds, descriptive statistics, and the participant recruitment configuration on Prolific. The results of the main study, including Prolific participant descriptive statistics and the objective and subjective measures, are provided in Chapter 5. These findings are subsequently discussed in Chapter 6, where we also reflect on the limitations and ethical considerations of our work and outline avenues for future research. Finally, Chapter 7 closes the thesis by summarizing its key contributions and insights.

2

Background and Related Work

This chapter provides the theoretical foundation and situates our contributions within the literature, drawing explicit connections between prior findings and the ideas we investigate. Section 2.1 begins with an overview of the challenges associated with compiler and interpreter errors, discussing their nature, common causes, and the difficulties developers encounter when diagnosing and resolving them. Section 2.2 then explores the emergence and role of large language models in software engineering, outlining their capabilities, limitations, how to evaluate them systematically, and potential applications for improving developer experience and productivity. We take a slight detour in Section 2.3 to discuss the concept of actionable insights and how it relates to teaching and learning in programming contexts, emphasizing the importance of clear, context-aware feedback for effective learning. Finally, Section 2.4 provides a comprehensive review of previous related work, examining approaches to error message design as well as research on using LLMs for error explanation and resolution.

2.1. Compiler and Interpreter Errors

Although the terms *compiler errors* and *interpreter errors* are sometimes used interchangeably, they refer to distinct types of feedback generated by different execution models in programming languages.

Compiler errors occur when a program fails to meet the syntactic or semantic requirements of a language during the compilation stage, preventing the creation of an executable artifact. Interpreter errors, in contrast, arise at run-time when an interpreter encounters an issue, such as a type mismatch, undefined variable, or unhandled exception, while executing the program.

However, the distinction is not always clear. For instance, Python, although typically described as an interpreted language, performs certain compilation steps (e.g., bytecode generation) before interpretation. Throughout this thesis, the term *(programming) error messages* will be used collectively to refer to both compiler and interpreter feedback, as many of the challenges in improving their clarity and usefulness are similar regardless of the underlying execution model.

2.1.1. Programming Paradigms

Programming languages can be broadly classified into different paradigms, such as procedural, functional, and object-oriented, each with its own design philosophy, abstractions, and typical application domains. Although these paradigms are often presented as distinct, many modern languages are in fact multi-paradigm. One such example is Python [17], which supports procedural, functional, and object-oriented constructs, whereas many other languages, such as Java or Kotlin, which are primarily considered object-oriented, have adopted functional features over time. Each paradigm emerged in response to specific challenges in software development, offering different ways of reasoning about and structuring programs. These differences influence not only how code is written and understood, but also the types of errors programmers tend to usually make, as well as the style in which these error messages are written. Below we summarize notable features of the most common programming paradigms.

- **Procedural programming** (e.g., C, Fortran, COBOL) describes computation as a sequence of explicit instructions that modify the state of the program. *Strengths:* Direct and intuitive for expressing step-by-step algorithms, similar to the way machines execute code. *Weaknesses:* Readability and maintainability can suffer in large codebases due to reliance on mutable state and control flow complexity. *Common mistakes:* Incorrect variable initialization, unintended state changes, improper control flow (e.g., misplaced loops or conditionals) and memory mismanagement in low-level contexts.

```c
#include <stdio.h>

int main() {
    int x = y + 5;
    printf("%d\n", x);
}
```

(a) Buggy C code snippet.

```
Main.c: In function ‘main’:
Main.c:4:13: error: ‘y’ undeclared (first use in this function)
    4 |     int x = y + 5;
      |             ^
Main.c:4:13: note: each undeclared identifier is reported only once
for each function it
```

(b) Associated error message.

Figure 2.1: Example of a small code snippet and its associated error in a procedural programming language, C.

- **Functional programming** (e.g., Haskell, F#, Lisp) treats computation as the evaluation of mathematical functions, emphasizing immutability, declarative expressions, and higher-order functions [32, 31]. *Strengths:* Strong suitability for parallel and concurrent processing, predictable behavior due to immutability, and expressive abstractions for transformations. *Weaknesses:* Steep learning curve for programmers accustomed to procedural thinking, verbose or esoteric syntax in some languages, and performance trade-offs in certain scenarios. *Common mistakes:* Type mismatches, misunderstanding of lazy evaluation, incorrect function composition, and difficulty reasoning about monads or other advanced abstractions.

```haskell
applyTwice f x = f (f x)
result = applyTwice double unknownVal
double n = n * 2
main = print result
```

(a) Buggy Haskell code snippet.

```
Main.hs:2:28: error: Variable not in scope: unknownVal
  |
2 | result = applyTwice double unknownVal
  |                            ^^^^^^^^^^
```

(b) Associated error message.

Figure 2.2: Example of a small code snippet and its associated error in a functional programming language, Haskell.

- **Object-oriented programming** (e.g., Java, C#, Ruby) organizes code around objects, state bundles (fields) and behavior (methods) often using principles such as encapsulation, inheritance, and polymorphism. *Strengths:* Enables modeling of real-world entities, supports the organization of large-scale systems, and encourages reuse of code through inheritance and composition. *Weaknesses:* Overuse of inheritance can lead to fragile hierarchies, while deep object coupling can reduce flexibility and testability. *Common mistakes:* Incorrect method overriding, violating encapsulation, misuse of inheritance instead of composition, and misunderstanding the object life cycle.

```java
public class Main {
    public static void main(String[] args) {
        int x = y + 10;
        System.out.println(x);
    }
}
```

(a) Buggy Java code snippet.

```
Main.java:3: error: cannot find symbol
        int x = y + 10;
                ^
  symbol:   variable y
  location: class Main
1 error
error: compilation failed
```

(b) Associated error message.

Figure 2.3: Example of a small code snippet and its associated error in an object-oriented programming language, Java.

- **Logic programming** (e.g., Prolog, Datalog, Mercury) is a declarative paradigm in which programs consist of facts and rules, and computation is performed by querying these rules to infer logical consequences. *Strengths:* Naturally suited for symbolic reasoning, knowledge representation, and solving constraint-based problems. Programs tend to be concise and highly expressive for theorem proving, and search. *Weaknesses:* Performance can be unpredictable due to the

search-based execution model. Programs may be less intuitive for developers accustomed to procedural or object-oriented approaches. *Common mistakes:* Infinite recursion due to poorly ordered rules, unintended variable bindings, and overly general queries that lead to inefficient backtracking or failure to terminate.

The paradigm in which a language operates strongly influences the nature and presentation of programming errors. For instance, statically typed functional languages often emit complex type inference errors that require understanding advanced type system concepts, while dynamically typed languages may generate runtime exceptions with limited contextual information. Similarly, procedural code may produce errors related to incorrect parameter passing or global state conflicts, whereas object-oriented code may fail due to misuse of inheritance or polymorphism.

Although this thesis focuses on improving error messages in Python, a multi-paradigm language with a predominantly procedural and object-oriented flavor, understanding the broader landscape of programming paradigms provides important context. The underlying paradigm shapes both the origin of errors and the way compilers or interpreters communicate them, which in turn influences how error messages can be made more effective.

2.1.2. Impact on Developer Experience

Compiler and interpreter errors have a direct impact on the developer experience. Unclear or verbose error messages can increase cognitive load, leading to frustration [12], reduced productivity, and a higher likelihood of introducing further mistakes (i.e., cascading errors).

For novice programmers, this problem is particularly important. Without prior mental models of how errors typically manifest, they may find it difficult to interpret technical jargon, and locate the source of the problem, thus often resorting to seeking any means of external help [25]. This can hinder learning and discourage continued participation in programming activities. In fact, among novice programmers, poor design of programming errors is often cited as a major barrier to learning, even being a reason why many learners drop out of computer science (or other technical) degrees and switch to other fields [70, 27]. Experienced developers, while often better equipped to navigate confusing diagnostics, still face productivity losses when error messages are incomplete, misleading, or overly generic.

In both cases, the quality of error reporting shapes not only the efficiency of debugging but also the overall usability of the programming environment. As a result, research into improving the clarity, accuracy, and contextual relevance of error messages has significant implications for developer satisfaction and effectiveness.

2.2. Large Language Models in Software Engineering

Since the public release of OpenAI's ChatGPT in November 2022, large language models have attracted substantial attention within the software engineering community. These machine learning models, trained on vast corpora of natural language and source code, demonstrate remarkable proficiency in understanding and generating human-like text across diverse domains. While initially celebrated for tasks such as summarization, sentiment analysis, question answering, and translation, their utility has expanded rapidly to encompass software engineering applications.

By 2023, LLMs were being evaluated for their ability to produce small code snippets and assist with auto-completion. By 2025, they had matured to the point where they could generate entire functional software systems, including SaaS applications, from high-level natural language descriptions (e.g., platforms such as Lovable, Replit, GitHub Spark, and others[1]). This growth has been accelerated by advances in model architectures, training methodologies, and integration with development tools, as well as by significant investment from major technology companies.

2.2.1. Overview of LLM Capabilities

Large language models have rapidly advanced in both research and practice, offering a wide range of capabilities relevant to software engineering [59, 29]. These can be grouped into four main areas:

[1] Lovable - https://lovable.dev/; Replit - https://replit.com/; GitHub Spark - https://github.com/features/spark

- **Code generation, completion, and repair**: LLMs can produce syntactically valid code from natural language descriptions, complete partial snippets with contextually appropriate content, and refine or repair code through iterative feedback. These abilities stem from large-scale training on open-source code and natural language, allowing them to capture both structural syntax and common patterns.
- **Code comprehension and explanation**: LLMs can translate code into clear, human-readable explanations, annotate APIs, and generate documentation. This capability is valuable for understanding legacy codebases, onboarding new developers to existing systems, and summarizing complex logic for easier maintenance.
- **Error diagnosis and test generation**: By interpreting compiler or runtime feedback, LLMs can identify probable causes of errors and suggest targeted fixes. They can also produce unit and integration tests to validate correctness, though these outputs still require human verification to ensure semantic accuracy.
- **Agentic behavior and cross-domain reasoning**: LLMs are capable of decomposing tasks, invoking external tools, and reasoning across different knowledge domains. This allows them to function as autonomous or semi-autonomous agents, integrating code generation with planning and tool-assisted problem solving.

These capabilities have driven the emergence of AI-powered developer tools, ranging from in-IDE coding assistants to autonomous multi-agent systems that can handle entire development workflows.

2.2.2. Limitations and Mitigation Strategies

Despite their impressive versatility, LLMs exhibit several limitations that affect their reliability and safe deployment in software engineering [59, 23]. Put shortly, while a lot of enthusiasm is warranted for the potential of LLMs (to continue) to revolutionize software development, arguably even more caution is warranted in their use, especially in production environments. Some of the most notable shortcomings include:

- **Context window constraints**: LLMs have a fixed limit on how much information they can consider at once. In long interactions or when working with large codebases, they may lose relevant details or produce inconsistent outputs. While techniques like chunking and retrieval-augmented generation (RAG) can help mitigate this, they introduce additional complexity and potential for errors. Recent advances in LLM architectures, such as the development of models with larger context windows, have continued to improve this situation, but practical limits still exist.
- **Hallucinations and factual errors**: LLMs can produce plausible-sounding but incorrect or entirely fabricated information. In software contexts, this can lead to faulty API calls, invalid syntax, use of non-existent functions or libraries, and other issues that can cause runtime errors or unexpected behavior. While they can generate code that appears syntactically correct, the underlying logic may be flawed or not aligned with the intended functionality. One can imagine that this is particularly problematic for novice programmers, or even experienced programmers who are not familiar with the specific domain or framework of the code being generated.
- **Semantic gaps and surface-level fluency**: While proficient in generating syntactically correct code, LLMs often lack deep semantic understanding, making them prone to introducing subtle logical flaws or overlooking domain-specific constraints. When coupled with small context windows or incomplete prompts, this can result in outputs that superficially resemble valid code but fail to meet the intended requirements or best practices.
- **Bias and knowledge staleness**: Because they are trained on large, static datasets, LLMs can inherit biases present in their training data and fail to incorporate the most recent knowledge or best practices without retraining or augmentation. This can lead to outputs that reflect outdated conventions, security vulnerabilities, or other issues that have since been addressed in the software engineering community.
- **Ethical and safety concerns**: The potential for LLMs to generate harmful, misleading, or otherwise inappropriate content raises ethical questions about their deployment in production systems. This includes risks of generating insecure code, propagating harmful stereotypes, or producing outputs that violate intellectual property rights. These concerns require careful consideration of

how LLMs are integrated into software development workflows, including the need for human oversight and validation.

- **Dependence on prompt quality**: The effectiveness of LLMs is highly dependent on the quality and specificity of the prompts they receive. While researchers at reputable institutions and companies are aiming to develop increasingly more robust and reliable LLMs, that can provide high-quality results even in the presence of noisy or incomplete prompts, the current state of the art (SoTA) models still require careful prompt design to elicit the desired behavior. Interestingly, a recently-published paper [52] highlighted that including one random fact within a prompt can greatly decrease the quality of the generated output, even for SoTA models, which is a clear indication that LLMs are still very sensitive to the input they receive.

To mitigate these limitations, practitioners have adopted several strategies:

- **Prompt engineering**: Carefully crafted prompts, often with explicit reasoning steps, can guide LLMs toward more accurate and contextually relevant outputs. Techniques such as chain-of-thought prompting, and few-shot learning, have been shown to improve performance on complex tasks by providing structured guidance and examples.
- **Model fine-tuning and domain adaptation**: Fine-tuning LLMs on domain-specific data or using specialized models can enhance their performance in particular contexts, such as software engineering. This approach allows the model to better understand the nuances of specific programming languages, frameworks, or application domains, improving its ability to generate relevant and accurate code. This is, of course, not without its own challenges, as it requires access to high-quality, domain-specific datasets and careful consideration of the model's training objectives. Even before the rise in popularity of LLMs, machine learning models were often trained on domain-specific data to improve their performance in specific tasks, such as natural language processing, computer vision, and much more.
- **Retrieval-augmented generation (RAG)**: Supplementing the model's input with retrieved, up-to-date, and domain-specific information helps ground responses and reduces the risk of hallucinations. This approach is also useful for providing highly-sensitive or context-specific information that may not be present in the model's training data. While not related directly to software engineering, companies within the legal or medical fields have been using RAG to provide up-to-date information to their LLMs, further improving the quality of results, while also mitigating privacy concerns by ensuring that sensitive information is not included in the model's training data. Naturally, this is also highly relevant to software engineering, where a company's private codebases, which were naturally not included in the training data of the model being used, can be used to provide up-to-date information to the LLM, allowing it to generate more relevant and accurate code, only in the context of that specific company.
- **Human-in-the-loop and iterative validation**: Involving developers in reviewing, testing, and refining model outputs ensures correctness, safety, and alignment with project goals. Human oversight is crucial for catching errors, validating assumptions, and ensuring that the generated code meets the intended requirements.

In the context of this thesis, focused on improving Python error messages with LLMs, these strengths and weaknesses are essential to acknowledge. They help define the scope, highlight the importance of prompt design, and justify the decision to limit empirical evaluation to Python rather than attempting to cover multiple programming paradigms at once. While we obviously did not explore all of the aforementioned strategies in this thesis, they nonetheless provided sufficient context for understanding the potential and limitations of LLMs, and therefore to addressing our research questions.

2.2.3. LLM Providers and Benchmarks

When designing and evaluating LLM-powered solutions, one of the most important prerequisites is to have a holistic view on what models to choose from. Currently, a wide range of providers offer pretrained language models, both for general-purpose use and for domain-specific applications. Programming has become one of the primary domains where such models are being adopted to support development workflows and accelerate research progress. Some providers have released model weights under open licenses, enabling their use for inference and further fine-tuning, thereby contributing to significant

innovation. Notable examples include Meta with the Llama family [21], Mistral AI with the Mistral models [35], and Alibaba Cloud with the Qwen series [33]. More recently, DeepSeek AI has attracted attention with the DeepSeek family, in particular for introducing fine-tuning methods based on reward modeling that reduce training costs [15]. In contrast, other providers restrict access to inference via APIs, without releasing the underlying weights. Noteworthy examples include OpenAI with the GPT models [47], Anthropic with the Claude family [3], and Google DeepMind with Gemini [63].

To assess which models are most suitable for software engineering tasks, systematic evaluation methods are needed to capture their capabilities and limitations. A number of benchmarks address code generation performance, typically through unit-test evaluation. Widely used Python-focused benchmarks include HumanEval [14] and EvalPlus [42], the latter focusing on the thoroughness of model evaluation. The MBPP benchmark [5] extends coverage with a larger set of problems. However, pure code synthesis from natural language does not capture the full range of developer interactions. To address this, LiveCodeBench [34] expands evaluation to a broader set of coding-related tasks, such as self-repair (assessing the ability to fix code using execution feedback) and test output prediction (measuring code comprehension). Yet, these benchmarks still focus on self-contained tasks, which may not fully reflect real-world development scenarios. For this reason, SWE-bench [36] evaluates models on resolving GitHub issues that involve multiple files and higher bug complexity, thereby moving closer to realistic software maintenance tasks. In addition, IFEval [74] has been proposed to specifically assess instruction-following capabilities, offering complementary insights into how well models align with user-provided prompts beyond code synthesis alone.

From the results of these benchmarks, a consistent pattern emerges: there is a substantial performance gap between SoTA closed models and open-weight models. In particular, the GPT and Claude families currently lead, largely attributed to their exposure to extensive, high-quality training data. Nevertheless, fine-tuning with curated datasets has been shown to substantially improve performance, whether to augment specific coding tasks or to better align models through instruction tuning [33, 22, 62].

Despite these advances, existing benchmarks face notable limitations. Data contamination, where evaluation examples overlap with training sets, remains a persistent concern, and current tasks still fall short of comprehensively measuring output quality, problem complexity, and real-world applicability. Most critically, current evaluations provide limited insight into how effectively models augment the developer experience. Beyond solving code synthesis tasks, models should be assessed on their ability to collaborate productively with developers. For instance, Kang et al. [37] highlight this need by introducing a technique that not only generates code fix patches but also produces intelligible explanations of the repair process.

2.2.4. Current Trends in Developer Experience

While recent advances have demonstrated remarkable capabilities of LLMs in software engineering, the idea of developers being replaced still belongs more to science fiction than reality. Instead, the emerging trend points towards collaboration, where developers and models work together to automate routine tasks and tackle more complex challenges. This augmentation is increasingly visible in the rise of agentic systems deployed within enterprise environments. Agents can be understood as autonomous systems that operate over extended periods, use diverse tools to accomplish complex tasks, and ground their actions in reasoning and environmental observations [2].

In software engineering, such agents are already finding their way into popular development environments, for example through integration with JetBrains IDEs[2]. The research community has pursued similar directions with systems such as SWE-agent [69], which autonomously interacts with developer tools, including compilers, debuggers, and version control systems to address real-world software engineering tasks. At present, agents typically perform the bulk of the work, with developers guiding, reviewing, and approving their outputs. This collaboration highlights a central challenge: errors and bug fixing remain inevitable, making it crucial that models are not only effective at their assigned tasks but also capable of interacting meaningfully with developers across varying backgrounds and expertise levels. In parallel, personalization has emerged as a key trend, with increasing emphasis on adapting models to individual developers [72].

[2] https://www.jetbrains.com/junie/

Personalization also extends beyond the human level to the task itself. As demand for agentic workflows grows, cost considerations have become a central concern [10]. One promising line of research advocates the use of smaller models, on the order of millions rather than billions of parameters, that can be fine-tuned for niche tasks and deployed as part of a fleet of specialized models. A practical example is JetBrains' Mellum, a 4-billion parameter model designed specifically for code completion and intended for local deployment within repositories [56]. This reflects a broader shift toward domain-specialized models, a trajectory that suggests future systems will not only be tailored to particular tasks but also increasingly to the needs of individual developers.

2.3. Actionable Insights and Teaching

Throughout this study, we understand *actionable insights* as feedback that is (1) clear and specific to what the learner is doing, (2) practical enough to act on without extra guesswork, and (3) timely, arriving right when a problem occurs. In programming, error messages are one of the most common forms of feedback. When they are vague, overloaded with jargon, or poorly placed, they add unnecessary mental effort and slow down both debugging and learning. When phrased clearly and placed in the right context, however, they can serve as small but powerful *teaching moments*: concise messages that help learners get back on track and build long-lasting mental models.

From a teaching perspective, actionable feedback aligns well with ideas in learning sciences [38]: *scaffolding* (providing just-enough support), *contingency* (adapting support to the learner's apparent need), and showing or hinting at the next correct step while reducing irrelevant detail. It also connects to *cognitive load* considerations: clearer, better-localized messages reduce extraneous load, allowing the learner to devote effort to the *germane* aspects of the task (i.e., diagnosis, fix, and reflection). For novices, who have not learned how to "read between the lines", these properties are especially important. For experts, succinct, non-intrusive explanations help maintain focus and avoid over-specification.

From pedagogy to design choices, we operationalize actionable insights through two message styles that map directly onto instructional strategies:

- **Pragmatic (task-oriented)**: concise, directive guidance that localizes the fault and proposes the next concrete step (e.g., "*Convert* x *to* int *before addition*"). This mirrors *procedural* guidance and is designed to minimize detours and reduce extraneous load.
- **Contingent (scaffolded)**: short, guided hints or questions that respond to likely misconceptions (e.g., "*Are you adding a string to an integer? If so, type cast one side.*"). This mirrors *contingent scaffolding*: support that theoretically should adjust to the learner and fade as understanding grows.

These styles complement the taxonomy of issues introduced earlier (Section 2.1) and the multi-paradigm context (Section 2.1.1). For example, runtime type errors in dynamically typed, imperative settings might benefit from highly localized, pragmatic guidance, whereas complex type-inference diagnostics in functional settings may call for scaffolding that reveals hidden assumptions before proposing a fix.

Framing error messages as actionable teaching interventions motivates the two goals of this study:

1. **Objective effectiveness** (RQ1): if messages are actionable, they should measurably improve *code-correction performance* (e.g., higher fix rates), independent of preference.
2. **Subjective perception** (RQ2): if messages reduce extraneous load and support understanding, participants should report higher *readability*, lower *perceived cognitive load*, and lower *perceived authoritativeness* (i.e., a more supportive, less prescriptive tone).

The *skill-adaptive* framing hypothesis follows directly: novices would benefit from slightly more structure and explication, while experts would benefit from brevity and minimal disruption.

Finally, although we do not adopt a specific educational framework verbatim, two concepts help orient our study: Bloom's taxonomy [39] and argumentation models. In Bloom's terms, error messages would function as micro-interventions that move learners from *Understand* (i.e., what failed) to *Apply* (i.e., how to fix) and, when needed, to *Analyze/Evaluate* (i.e., why it failed, while seeking alternatives) [4]. From the argumentation point of view, lightweight structures, such as Toulmin's *claim-evidence-warrant* [6] and the compact *Claim-Evidence-Reasoning (CER)* [44], may offer a logical way to name the fault,

point to the line, and give a plain-language reason, improving clarity without sounding too prescriptive. In particular, Toulmin's model informed our contingent prompt template, which we cover in detail within Section 3.3.

2.4. Related Work

This section places our study at the intersection of three areas of research: (1) foundational work on the design and impact of programming error messages (clarity, tone, timing, context), (2) recent efforts that use large language models (LLMs) to explain or rephrase errors, and (3) human-computer interaction (HCI). We first summarize core design principles and error patterns that motivate our constructs (readability, perceived cognitive load, perceived authoritativeness). We then review LLM-based approaches, highlighting where evidence converges or conflicts, and how these insights inform our skill-adaptive intervention.

2.4.1. Foundations: What makes an error message effective?

A comprehensive review by Becker et al. [9] maps five decades of research on PEMs, revealing several noteworthy themes relevant for PEM improvements: improve *readability* and reduce *extraneous cognitive load*, provide *context* and concrete *hints/examples*, adopt a *supportive tone*, use *logical structure* in explanations, and surface diagnostics at the *right time*. These themes directly motivate some of the constructs we measure in our study, and the error message properties we target (i.e., localization, plain language, actionable next steps).

Complementing design guidance, several studies examine the *distribution* and *types* of errors developers encounter. One study [51] shows that error frequencies in Python and Java follow a Zipf-Mandelbrot-like distribution, with a small set of error types (e.g., `SyntaxError`, `NameError` in the case of Python) dominating. This distribution suggests that targeted improvements to high-frequency cases can yield the most impact. Work by Shirafuji et al. [57] further differentiates *novice* and *expert* error profiles: novices more often lack knowledge of API existence/usage/specification, whereas experts more often misread problems or overlook constraints, evidence that error message design may need to adapt to skill. Beyond syntax, Ettles et al. [18] and Hristova et al. [30] catalog logical/semantic and common Java errors, respectively, highlighting that many failures reflect misconceptions rather than mere syntax violations, implying that brief, well-placed *why*-style cues (not only *what*) within error messages can improve understanding.

2.4.2. LLMs for Explaining and Rephrasing Errors

Recent systems explore LLMs as on-demand "explainers" or "rephrasers" of compiler/interpreter output. Taylor et al. [61] integrated an LLM into the `dcc` C compiler to help CS1/CS2[3] learners, reporting conceptually accurate explanations in most compile-time and many run-time cases, while noting issues with adhering to prompt instructions (e.g., revealing solutions when being explicitly told not to) and weaker run-time quality. Within another study, Leinonen et al. [41] showed that Codex-generated enhancements can be more novice-friendly than standard Python messages but suffer from incomplete or incorrect fix suggestions (e.g., only a subset included fixes, and even less were correct). Their experiments also suggested that a low temperature of 0 improved the consistency of high-quality outputs.

Evidence is mixed on whether LLM-enhanced messages *improve objective performance*. A 2024 study by Santos et al. [54] found that LLM-enhanced PEMs often underperform both standard and expert-crafted variants on several tasks, raising questions about when and how to deploy LLMs. In contrast, Salmon et al. [53] reported that GPT-3.5 can produce useful *code-only* explanations (without the original error text), though correctness gains mostly came from trimming verbosity via one-shot/fine-tune rather than large accuracy jumps. Another study by Santos et al. [55] found that including the *full source context* together with the PEM substantially improved clarity and fix specificity with GPT-4, proving that prompting matters and that one should always provide code context to LLMs. Complementing these, Wang et al. [66] conducted a large-scale randomized controlled study (RCT) across six Python message variants (from raw interpreter to GPT-4 explanations and forum links), finding that GPT-generated messages can reduce repeated runs and fix attempts, indicating practical benefits in authentic settings.

[3] CS1/CS2 are acronyms which typically refer to introductory programming courses for Computer Science university degrees.

2.4.3. Beyond Compilers: LLMs for Developer Diagnostics

Beyond compiler and runtime errors, LLMs have also been applied to related diagnostics, including CI/CD failure logs. A recent large-scale empirical study proposed the first taxonomy of GitHub Actions (GHA) problems [71]. Partly in response to that, Toledo et al. [65] use LLMs to explain GHA failures. The authors report that developers, especially less experienced ones, rate LLM explanations of GHA failures positively on correctness and clarity for smaller logs, while experienced developers prefer more concise summaries for complex scenarios. This reinforces a central premise of our own work: explanations should be *audience-aware*, and brevity versus scaffolding is a skill-sensitive trade-off.

LogPrompt, a system by Liu et al. [43], leveraged several prompting techniques to improve LLM performance on automated log analysis tasks, such as anomaly detection, log parsing and log interpretation. While LogPrompt is not directly related to error messages, it highlights the potential of LLMs to assist with diagnostics in software engineering contexts. The different prompt engineering techniques used for LogPrompt, are also relevant to our work on improving error messages, and have partly informed our initial exploration of prompting methods.

2.4.4. Synthesis and Positioning

Across these threads, three gaps emerge. First, while readability and cognitive load are longstanding concerns, few evaluations thoroughly distinguish *objective correction outcomes* from *subjective perceptions* in a single controlled design. Second, despite suggestive evidence that novices and experts need different levels of support, *skill-adaptive* rephrasings remain underexplored. Third, results for LLM-enhanced messages vary: some studies show improvements, others report null or negative effects, pointing to the importance of prompt design, message structure, and appropriate use of code context.

Our study addresses these gaps by (1) evaluating *objective* code-correction performance and *subjective* perceptions (i.e., readability, perceived cognitive load, perceived authoritativeness) within one experiment, (2) explicitly testing *skill-adaptive* styles, *pragmatic* (brief, fix-oriented) and *contingent* (detailed, “intent-aware” scaffolds), and (3) crafting prompts containing both the original interpreter message and source code context to minimize mis-localization and improve specificity. In doing so, we align with prior design advice (clarity, tone, timing, localized context), attempt to reconcile conflicting findings around LLM effectiveness by standardizing inputs and evaluation criteria, and extend the literature with a targeted look at novices vs. experts.

3

Research Methodology

This chapter describes the methodology used to investigate and answer the research questions in Section 1.2. We proceed in four stages. In an initial pilot study, we develop an objective, task-aligned assessment of Python proficiency (Section 3.1). A second pilot study (Section 3.2), then curates "the Fantastic Four", four buggy Python code snippets selected for realism and calibrated difficulty, for use in the main study. Third, we describe the design of our prompt templates and motivate the choice of the LLM to generate rephrased error messages (Section 3.3). Finally, in Section 3.4, we present the Prolific-based web application that operationalizes the main crowdsourced study. To aid the reader, Figure 3.1 illustrates the project approach, showing how both pilot studies and the main experiment were conducted to address our research questions.

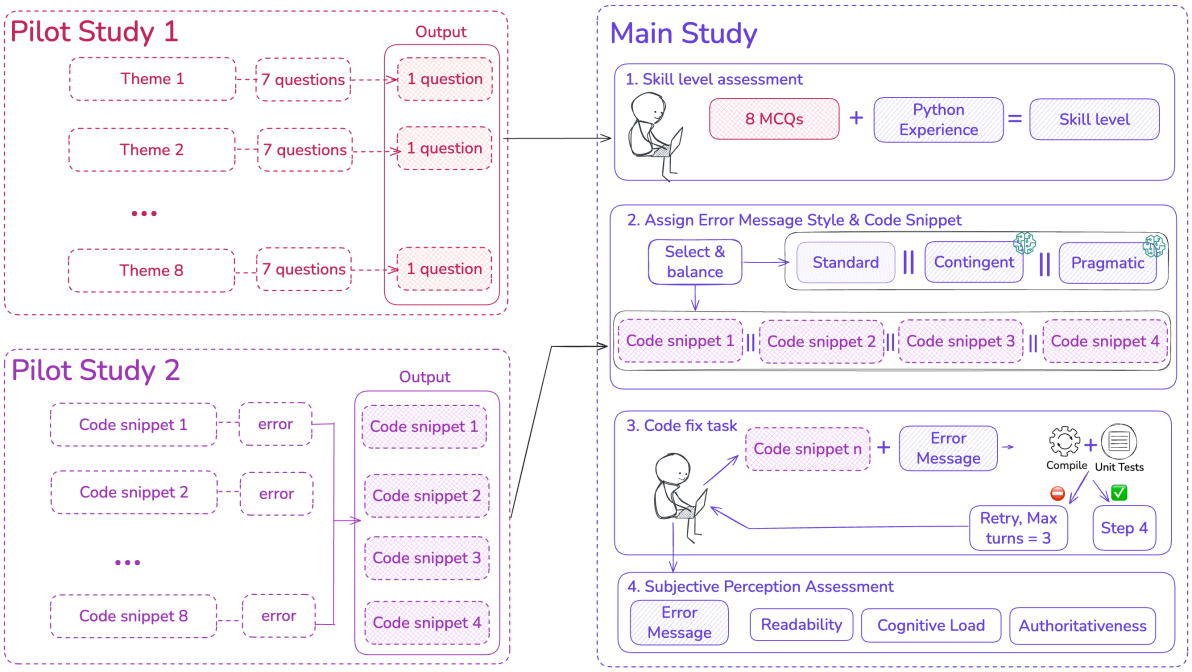


Figure 3.1: Project approach overview, including pilot studies and the main study conducted to address the research questions.

3.1. Python Proficiency Level Assessment

To meaningfully evaluate noteworthy effects across skill levels (RQ1–RQ2), we first require a reliable measure of Python proficiency that reflects the kinds of abilities involved in reading, understanding and fixing errors. This section motivates and details the design and results of our assessment instrument.

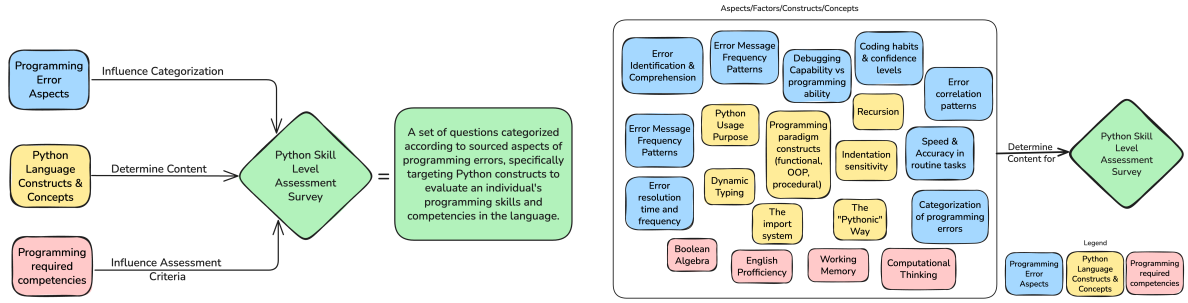
3.1.1. Motivation

Self-reports (e.g., Likert ratings of expertise or years of experience) are convenient, but known to suffer from bias and weak alignment with specific competencies such as debugging or error interpretation. Because our study focuses on how people understand error messages and repair faults, we sought a task-aligned assessment that emphasizes those skills. Although no proficiency measure is perfect, an instrument targeted at error comprehension, fault localization, and modifications should more effectively enable the skill-level classification required for our analyses.

Multiple formats (MCQ, fill-in-the-blank, coding tasks, open-ended responses) are generally viable means to assess skills. However, given the scale of the planned main crowd-sourced study and the need to control cognitive load before the main tasks, we opted for a multiple-choice format with carefully designed items that assess relevant sub-skills.

3.1.2. Approach

We conducted a literature-informed analysis of programming proficiency measures and of the anatomy of Python errors (e.g., syntax, semantics, runtime behavior) to derive assessment themes aligned with our goals. We then authored a bank of 56 multiple-choice items: 7 items across 8 themes relevant to debugging. A high-level visualization of this approach is presented within Figure 3.2. The themes are:



(a) High-level view of constructs influencing the content of the survey. **(b)** In-depth view of constructs influencing the content of the survey.

Figure 3.2: Visualization of the approach to create the multiple-choice questions for the Python skill-level assessment survey.

- **General Programming Error Understanding**: We recognize that a flawless programming language, one that would be ideal for every programming requirement, does not exist. Consequently, it is not unusual for programmers to use multiple programming languages professionally or to transition between them over the course of their careers. For this reason, we crafted certain questions aimed at evaluating a fundamental comprehension of programming errors, irrespective of the language utilized, as numerous concepts are shared across different languages.
- **Python-Specific Error Understanding**: Beyond grasping general programming errors, we crafted questions specifically focused on Python, including its syntax and semantics. These questions aim to cover Python-specific elements, which might not be common across other languages, such as applying indentation to define code blocks and using unique keywords and constructs inherent to Python.
- **Code Reading & Understanding**: Effective debugging requires code comprehension at both syntactic and semantic levels. Consequently, we included items that assess this competence, explicitly accounting for Python's dynamic typing, which can increase ambiguity and reduce comprehension, particularly in the case of novices.
- **Error Message Comprehension**: Error-message comprehension is foundational to debugging and typically the first actionable step after code execution. We therefore included items that assess the ability to parse Python diagnostics (i.e., exception type, traceback and offending line(s)) and map them to the underlying fault.
- **Error Identification**: Following message interpretation, the next step is fault localization in the source code. This step is demanding, particularly for novices, because it requires integrating code comprehension with diagnostic cues. We therefore included items that assess the ability to identify errors in Python code.

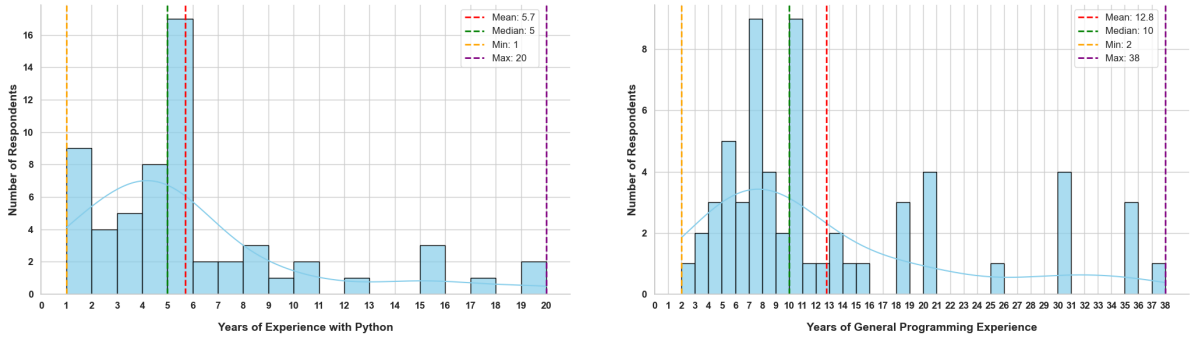
- **Error Resolution**: After fault localization, the task shifts to repair selection and implementation. Although related, identification and resolution are distinct: recognizing the fault does not guarantee knowledge of an effective remedy. As such, we included items that present the fault and its cause and require participants to choose the correct fix.
- **Natural Language Scenarios**: We also incorporated natural-language scenarios in which participants receive a problem description without any accompanying code, and must identify the most likely cause of the defect. This theme approximates real-world conditions where developers interpret imperfect specifications and reason about unfamiliar situations.
- **Miscellaneous**: We additionally included multiple-choice items that span multiple categories, are more difficult, and which would hopefully be more reflective of compact real-word scenarios.

Administering all 56 items to participants would introduce significant cognitive load. We therefore designed our pilot study so that each participant received a 16-item subset, with 2 randomly sampled items per theme. Theme order, item order within themes, and option order were fully randomized to mitigate ordering and learning effects. The survey was implemented in Qualtrics[1] and distributed via convenience sampling (i.e., LinkedIn, the CHI Nederland community, authors' networks) to achieve a balanced mix of experience levels.

Alongside the instrument, participants reported their self-assessed Python proficiency level via Dreyfus' model [11] of skill-acquisition[2] and years of experience (Python, as well as general programming) to serve as comparative baselines. Finally, participants estimated how many of the 16 questions they believed they answered correctly. Prior work [19] suggests self-evaluations can correlate with work quality in crowdsourcing contexts, and we explored whether this held here as well, and would be beneficial for us in our main study. In the end, we decided to drop this method of participant pre-selection for the Prolific study, as we had enough measures of quality control and screening. The resulting item statistics from this pilot study informed the selection of a high-discriminative subset of questions to be used in the main study (see Section 3.4).

3.1.3. Pilot Study Results

For this pilot study, 78 participants started and 60 completed the survey, yielding a completion rate of 76.92%. The median response time was 16 minutes; we do not report the mean because response times were skewed by the 7-day completion window. Distributions of self-reported years of experience with Python and with programming in general are shown in Figure 3.3.



(a) Distribution of self-reported years of experience with Python (*M*= 5.7, *SD*= 4.55).

(b) Distribution of self-reported years of experience with programming in general (*M*= 12.76, *SD*= 9.60).

Figure 3.3: Self-reported years of experience with Python and programming in general.

Participants answered on average 10.78 of 16 items correctly (67.37% accuracy). When asked to estimate their own accuracy, the mean estimate was 12.2 out of 16 (76.25%). The Spearman correlation between self-estimated and actual accuracy was $r \approx 0.35$ with $p \approx 0.006$, indicating a moderate positive

[1] https://www.qualtrics.com/

[2] Interested readers can review Table A.1 which lists the six-level self-rating of Python experience that we used verbatim in the study, which was informed by Dreyfus' skill-acquisition model.

association. For further exploratory analysis and reference, Table 3.1 lists the full set of Pearson/Spearman correlations examined in this survey.

Variable A	Variable B	correlation (method)	p-value
Python YoE	General programming YoE	0.55^{P}	0.00001
Python YoE	Dreyfus level	0.52^{S}	0.00003
Python YoE	Self-estimated accuracy	0.24^{S}	0.00655
Python YoE	Actual accuracy	0.28^{P}	0.01527
Dreyfus level	Self-estimated accuracy	0.24^{S}	0.00655
General programming YoE	Actual accuracy	0.06^{P}	0.62229
General programming YoE	Self-estimated accuracy	0.03^{S}	0.82473
Actual accuracy	Self-estimated accuracy	0.35^{S}	0.00671

Table 3.1: Correlations examined in the Python skill-assessment survey. The correlation column also indicates the test used: $^{\mathrm{P}}$ for Pearson, $^{\mathrm{S}}$ for Spearman; Python YoE = self-reported years of experience with Python; General Programming YoE = self-reported years of general programming experience; Dreyfus level = Ordinal scale (1 = no experience, 6 = significant experience); accuracy = number of questions answered correctly, on a (0-16) scale.

For extracting the best-performing questions, we calculated the point-biserial correlation[3] between each question and the total score, as well as the difficulty of that question. When using an acceptable difficulty value range of 0.2 to 0.9 for each question, while keeping the best performing question within each block / topic, and after removing some questions that held little discriminative power, we selected the questions listed in Table 3.2 to be used in the skill assessment part of our main Prolific study.

Question ID	Difficulty	Discriminative Power (r_{pb})	p-value
Q4.6	0.82	0.50	0.04224
Q5.2	0.83	0.65	0.00351
Q6.5	0.84	0.60	0.00631
Q7.1	0.88	0.81	0.00008
Q7.2	0.56	0.51	0.04598
Q8.2	0.78	0.40	0.10440
Q9.5	0.76	0.30	0.24669
Q10.1	0.87	0.45	0.08909

Table 3.2: Difficulty, discriminative power, and p-value for the final selected questions of the Python skill-assessment survey.

The values in the discriminative power column represent the point-biserial correlation between each question and the total test score, essentially reflecting how well an item can distinguish between high- and low-performing participants[4]. Higher values correspond to stronger discriminative power. For interpretation, we use the following guideline:

- **0.00–0.19**: Weak discriminative power; item does not properly separate high and low scorers.
- **0.20–0.29**: Fair discriminative power; item shows some separation ability.
- **0.30–0.59**: Good discriminative power; item effectively distinguishes between groups.
- $\geq$**0.60**: Excellent discriminative power; item very effectively distinguishes between groups.

Negative correlations may technically also occur (i.e., values are in the $[-1, 1]$ interval), meaning that lower scorers are more likely than higher scorers to answer the question correctly. Such cases typically suggest that the item is poorly constructed, however, we generally did not encounter such occurrences,

[3] We did not use item response theory (IRT) because our sample size was too small for stable item estimates. IRT usually needs many more responses per item, so the point-biserial measure within Classical Test Theory (CTT) was more appropriate.

[4] The value for r_{pb} should not be interpreted by itself, as one must also look at the corresponding p-value, which indicates the probability of observing the calculated r_{pb} value if the null hypothesis of no correlation is true. When $p < 0.05$, it suggests that the correlation is statistically significant and unlikely due to chance. In our case, most items had statistically significant positive correlations, however, Q8.2, Q9.5, and Q10.1 remain an exception. Since we did not have better alternative options for those respective item categories/themes, we ultimately decided to keep them in for the final assessment.

and none of the final set of questions exhibited this. The values in the difficulty column capture the proportion of participants who answered an item correctly: values near 1 indicate easier items, whereas values near 0 denote more difficult ones.

3.2. Selecting Candidate Code Snippets and Errors

The main study requires buggy snippets that are realistic, diverse, and appropriately challenging for novices and experts alike. This section describes how we constructed and reduced an initial pool of candidate options to a final set of four representative Python code snippets, and their respective PEMs.

3.2.1. Motivation

Given the vast space of possible Python faults, an exhaustive coverage is inherently impossible. Our objective was a pragmatic yet informed selection: snippets that (1) trigger commonly encountered Python error types, (2) vary in presentation and underlying fault cause to avoid bias toward any single pattern, and (3) sit in a "Goldilocks"[5] difficulty band, neither trivial nor unreasonably difficult, for our target mixed-skill population. We discuss potential limitations and generalizability about the approach that we used to source these buggy code snippets in Section 6.2.

3.2.2. Approach

Building on the literature review supporting Section 3.1, we targeted high-incidence error families (e.g., `SyntaxError`, `TypeError`, `NameError`) and used an LLM to generate candidate synthetic snippets that elicit these errors. The generation prompt that we used enforced the following constraints:

- **Self-contained**: Single-file Python programs relying only on the standard library.
- **Runnable structure**: Optional class and/or `main` function, which would be executable as a standalone script.
- **Balanced complexity**: ≤ 60 lines, at least 3-5 functions across class and standalone definitions, low cyclomatic complexity, and concise Python docstrings to make function intent explicit.

We used OpenAI's `GPT-4o-mini-high` to generate eight candidate snippets. Each was manually validated to ensure (1) it triggers the targeted error, (2) it complies with prompt constraints, and (3) it contains only a single fault to avoid cascading failures [8] when participants attempt program repairs. If any violations were found, we manually addressed them or re-generated snippets accordingly.

We then ran a pilot study in which participants rated each snippet on (a) perceived difficulty of understanding the code, (b) perceived difficulty of fixing, (c) mental demand of reading the error message, and (d) helpfulness of the error message, using 5-point Likert scales. Table 3.3 lists the exact questions that were used in the pilot study, which were designed to capture these four dimensions of interest.

Construct assessed	Survey item
Code difficulty	"*This code snippet is difficult to understand.*"
Fix difficulty	"*I would find it challenging to resolve the issue in this code snippet.*"
Error mental demand	"*Reading this error message feels mentally demanding.*"
Error usefulness	"*This error message is useful for identifying the problem.*"

Table 3.3: Mapping between the pilot study's survey dimensions and their corresponding items.

From this data we sensibly selected four snippets, i.e., the "Fantastic Four", that collectively span error types and difficulty while remaining suitable for our audience. For brevity, we present only one of the final 4 chosen snippets in Listing 3.1, and leave the interested reader to consult Appendix B, which lists the chosen four snippets, alongside their respective error messages, which were created by the standard Python interpreter when executing the code snippets as standalone files (i.e., `python <file_name>`). Detailed results of the analysis for this is reported within Section 3.2.3.

[5] "Goldilocks" denotes a just-right range, neither too easy nor too hard, inspired from the English fairy tale "*Goldilocks and the Three Bears*" (first published by Robert Southey, 1837).

```python
import random


class UserData:
    """Represents user data with a name and a list of scores."""

    def __init__(self, name, scores):
        self.name = name
        self.scores = scores

    def top_score(self):
        """Returns the highest score."""
        return maximum(self.scores) if self.scores else 0

    def add_score(self, score):
        self.scores.append(score)


def summarize_scores(users):
    return {u.name: u.top_score() for u in users}


if __name__ == '__main__':
    """Main routine to generate user data, summarize scores, and print the results."""
    users = [UserData(f"user_{i+1}", [random.randint(0, 100) for _ in range(random.randint
        (2, 5))]) for i in range(4)]
    summary = summarize_scores(users)
    for name, score in summary.items():
        print(f"{name}: {score:.2f}")
```

Listing 3.1: One of the chosen four buggy Python code snippet, which when executed as a standalone file, will trigger a `NameError` on line 13 of the code listing, due to the undefined variable `maximum`.

3.2.3. Pilot Study Results

In this pilot study, participants were recruited through both convenience sampling and Prolific, yielding a total of 20 valid responses (one additional participant was excluded for failing the attention check). Figure 3.4 presents the mean ratings for each candidate code snippet and error message across the four questions listed in Table 3.3. The relationships between these questions are further explored through a correlation matrix, shown in Figure 3.5. As expected, perceived fix difficulty positively correlates with error-message mental load. On the other hand, error message usefulness decreases as cognitive load and perceived difficulty rise. These patterns suggest that participants interpreted the questions as intended and that the selected candidate code snippets and messages were appropriate for the pilot.

From a brief glance alone, the snippet IDs reported in the mean ratings figure (3.4) do not convey much meaning by themselves as one cannot immediately see the underlying code and error for which the values are applicable. Establishing the connection between this figure and what was chosen is therefore challenging. For this reason, we provide the final four selected code snippets in Appendix B. For readers interested in a more detailed view, one of our public GitHub repository [45] hosts the full analysis scripts together with all candidate Python code snippets and their corresponding PEMs.

3.3. Prompt Templates and LLM Settings

This section describes how we designed the prompts that rephrase Python errors into more usable messages and how we selected the LLM that executes these prompts under our constraints. This represents one of the main contributions of our research, and is in effect a key element of our framework for generating skill-adaptive error messages.

3.3.1. Motivation

The quality of rephrased messages, and thus our ability to improve comprehension and repair, relies on (1) prompts that elicit readable, precise, and actionable feedback, while reducing cognitive load, and (2) an LLM that reliably follows instructions, handles code context, and produces concise, readable text, which incurs low cognitive load. Given LLMs' prompt sensitivity on small and large models alike [75, 13], prompts and model must be designed and evaluated jointly.

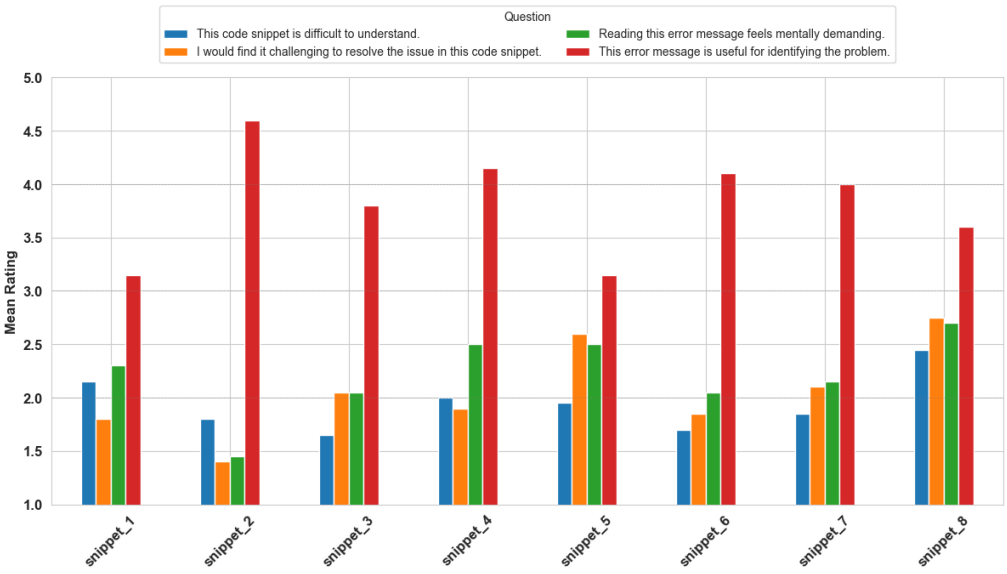


Figure 3.4: Mean ratings for each candidate code snippet and error message across all four questions used in the pilot study.

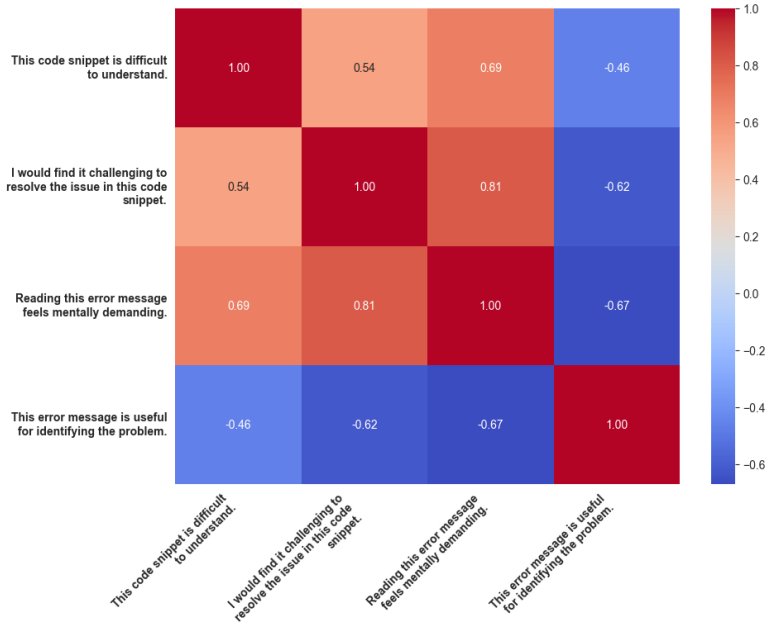


Figure 3.5: Correlation matrix between the four questions used in the Fantastic Four pilot study for all candidate code snippets.

3.3.2. Approach

We collected design principles from prompt-engineering best practices and analyses of Python diagnostics. Target properties of the rephrased messages were: (1) high readability (i.e., plain language, short sentences where appropriate), (2) low cognitive load (i.e., minimal jargon, explicit reference to the offending line), (3) precise localization cues, and (4) grammatical correctness. In line with our intent to provide skill-adaptive error messages, we created two closely related templates for rephrasing error messages: *pragmatic* (succinct, fix-oriented guidance) and *contingent* (scaffolded, explanatory feedback). In relation to our research questions, we hypothesize that pragmatic messages better suit advanced users with established mental models, whereas contingent messages better support novices who benefit from additional context and guidance. This scaffolding is intended to reduce jargon-driven overload and to make the messaging tone more empathetic to learners. Both pragmatic and contingent prompt templates that we used are referenced verbatim within Listings 3.2 and 3.3, respectively.

```
INSTRUCTION:
You are an assistant helping a Python programmer by explaining the error, in a clear and concise way.

CONTEXT:
The programmer's code and error message are below:
---
CODE:
```python
{code}
```
ERROR MESSAGE:
```
{error}
```

TASK:
1. Identify the cause of the error and the relevant line number from the code snippet. DO NOT FETCH THE LINE NUMBER FROM THE ERROR MESSAGE.
2. Write exactly one paragraph (around 20-25 words or less) that:
    - Begins with "**<ExceptionType>** at **line <line>**:"
    - Briefly states the cause and hints at a fix.
    - Focuses on providing helpful actionable insights, without directly giving the corrected code.

FORMAT:
You may use markdown for emphasis, but do NOT include lists or code fences.

NOW WRITE YOUR RESPONSE:
```

Listing 3.2: The Pragmatic error message prompt template.

```
INSTRUCTION:
Guide the Python programmer to resolve their error in 3-5 supportive sentences. Focus on actionable steps and encouragement,
while avoiding an authoritative tone.

CONTEXT:
The programmer's code and error message are below:
---
CODE:
```python
{code}
```
ERROR MESSAGE:
```
{error}
```

TASK:
1. Identify the cause of the error and the relevant line number from the code snippet. DO NOT FETCH THE LINE NUMBER FROM THE ERROR MESSAGE.
2. Then write 3-5 sentences that:
    - Begin your response with "**<ExceptionType>** at **line <line>**:" on the first line.
    - Then confirm the likely intent or goal (e.g. "Did you mean to ...?") - this is the claim, showing you understand what the programmer was trying to do.
    - Only if useful, mention whether this error is common or if there are exceptions.
    - Only if relevant, note whether there are situations where the error might occur differently.

FORMAT:
You may use markdown for emphasis, but do NOT include lists or code fences.

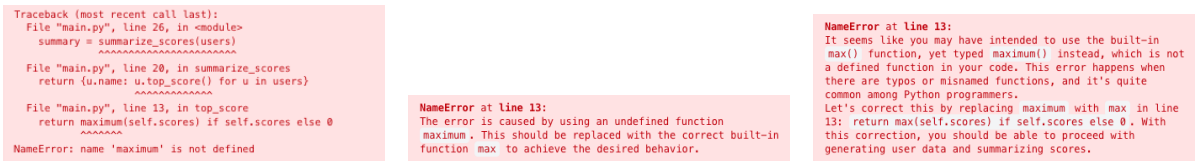
NOW WRITE YOUR RESPONSE:
```

Listing 3.3: The Contingent error message prompt template.

Because code context is crucial, each prompt includes (1) the exact snippet that produced the error and (2) the interpreter message/traceback, alongside our task instructions. We also saw that prepending the line numbers for the code snippet and the traceback to the prompt improves the model's ability

to reference them correctly in the rephrased message. Using the Python standard interpreter error message and the snippet's line numbers as ground truth within our prompts minimizes the risk of citing the wrong line in the LLM-rephrased message.

We formatted prompts in a mix of Markdown and plain text for clarity and consistency with common prompt engineering practices. However, since prior work [28] suggests formatting effects are generally secondary to content and depend on task and model, we fine-tuned our templates to the specific model and use case until finding something that works generally well. For better visualization of how these error message styles differ, Figure 3.6 illustrates all three examples which would be shown when running the code from Listing 3.1.



(a) Standard interpreter error message. **(b)** Pragmatic programming error message. **(c)** Contingent programming error message.

Figure 3.6: Examples of the standard, pragmatic, and contingent programming error messages styles, which would be shown when running the code in Listing 3.1. The pragmatic error message was generated using the template shown in Listing 3.2, and the contingent error message using the template in Listing 3.3.

In parallel, we tested open-source LLMs that could run on a self-hosted VM, follow instructions reliably, perform well on relevant code-related benchmarks, and handle enough context to include medium-length code along with its standard Python error message. Consequently, we reviewed and selected the `llama-3.1-8B-Instruct` model for generating the rephrased messages[6], balancing performance on the IFEval and HumanEval benchmarks, with single-machine deployment feasibility. In terms of other relevant settings, we set the temperature to 0 and used zero-shot prompting. We were informed of this optimal settings for our case by prior work [41] and our validation tests. A zero temperature maximized stability and reproducibility, while pilot runs showed that one- and few-shot prompts biased outputs toward the provided examples and reduced generalization.

3.4. Prolific Web Application Design

The main study was deployed as a custom web application integrated with Prolific for recruitment and eligibility enforcement. This section describes the system architecture, participant workflow, and task logic of the application. To aid the reader, Figure 3.7 illustrates the main participant-facing user interfaces. Additionally, Appendix C provides a description of the full-stack system and the key design choices.

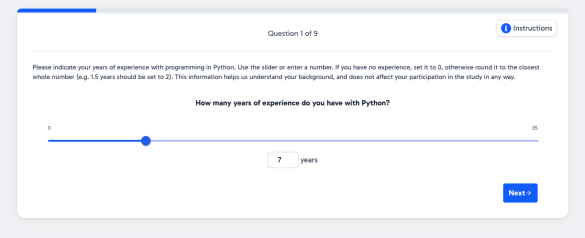
3.4.1. Motivation

To answer the research questions at scale with a diverse participant pool, we used Prolific solely for recruitment, prescreening, and compensation management. Prolific enables targeted screeners, prevents duplicate participation, and standardizes inclusion criteria, thereby improving external validity. After passing eligibility checks on Prolific, participants were redirected to our custom web application, independent of Prolific, where all tasks were administered and all responses were recorded. Prolific thus facilitated sampling and eligibility enforcement (e.g., country, age, prior exposure), while the study itself ran entirely on our platform and was responsible for the task logic, data collection, and participant experience.

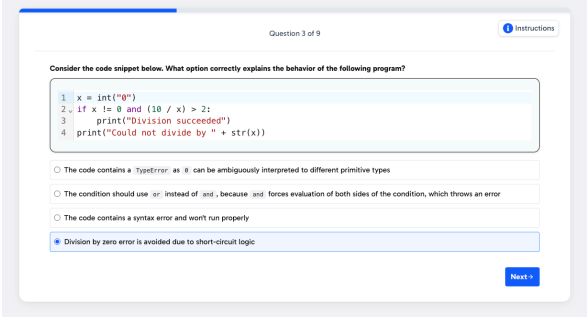
3.4.2. Approach

We implemented a full-stack application with Next.js/React on the frontend and FastAPI on the backend, persisting data in PostgreSQL. Participants arrive from Prolific with their unique Prolific ID (used as an URL parameter) for enforcing one completion per user and to align records across tasks. The UI prioritizes clarity (e.g., concise instructions, consistent layouts) and familiarity (i.e., code editor re-

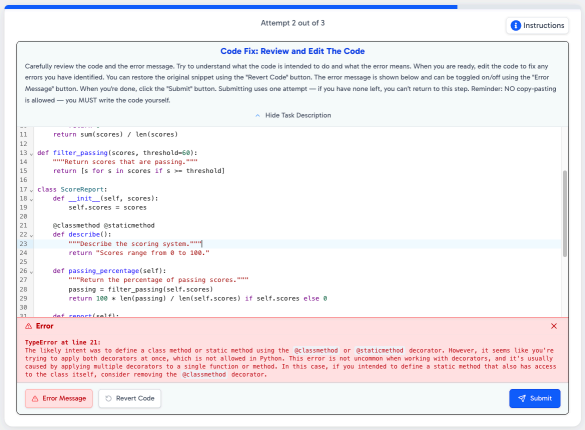
[6] To reiterate and avoid confusion: the candidate buggy snippets in the prior step (Section 3.2) were generated with `GPT-4o-mini-high` for efficiency, whereas all participant-facing rephrasings in the main study used `llama-3.1-8B-Instruct`.



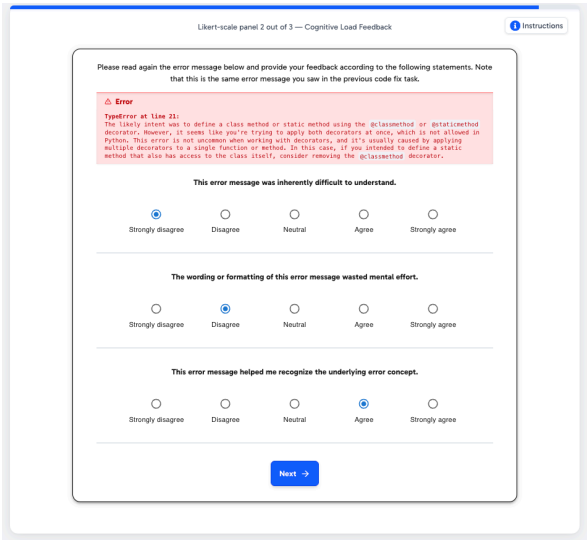
(a) Skill assessment screener: self-reported Python years-of-experience slider component used in skill grouping.



(b) Skill assessment screener: multiple-choice component used in skill grouping.



(c) Embedded code editor used for debugging tasks with the assigned error message style. The error message in this picture belongs to the contingent error message style.



(d) Post-task user feedback form. Participants need to provide their impressions on three different panels: (1) readability, (2) cognitive load, (3) authoritativeness.

Figure 3.7: User interfaces from the Prolific web application used in the study: (a) Python YoE skill screener, (b) MCQ assessment, (c) code editor interface, and (d) post code fix task subjective ratings feedback.

sembling common IDEs, such as PyCharm[7] or VSCode[8] with Python syntax highlighting and non-AI rule-based inline completion).

After providing their consent to our terms and means of collecting data, participants must review interactive instructions and a sandbox interface for each task in the survey before proceeding. The survey instructions can be accessed at any time later during the actual study. We log integrity signals (e.g., copy/paste/cut, window blur, tab switching) for manual review. Submissions exhibiting clear evidence of low-effort (e.g., random typing, repeated identical code fix submissions) or external assistance are rejected, as disclosed upfront. Participants can withdraw at any time, and their data is anonymized and used solely for this research. Once participants go through the consent and instructions, they proceed to the main study which consists of three sequential parts, as outlined below:

[7] https://www.jetbrains.com/pycharm/
[8] https://code.visualstudio.com/

1. **Python Skill Assessment:** Participants report their years of Python experience using a slider (see Figure 3.7a), and answer 8 MCQs (see Figure 3.7b), as identified by the pilot study described in Section 3.1. Items and options are randomized to avoid ordering effects, whereas backtracking is disabled to prevent answer revision based on later items. Using the assessment responses and years of experience, the backend logic of the application assigns participants a skill level. The exact skill-assignment logic is reported within Table 3.4.
2. **Code Fix Task:** After determining the participant's skill level, the assignment logic then selects one of the four curated snippets and one error-message condition (i.e., standard, pragmatic, or contingent). Assignment logic maintains both global balance (across all participants) and local balance (across snippets within message style). Participants attempt to repair the snippet in an embedded editor (see Figure 3.7c), which they had gotten familiar with during the survey instructions. A submission is considered correct if it removes the targeted error and passes a hidden test suite, which checks for semantic correctness. Participants were allowed up to three submission attempts in the code fix task. This cap was motivated by Wang et al.'s RCT [66] showing that LLM-enhanced PEMs reduce re-runs, with students repeating errors 23.1% less often and resolving them in 34.8% fewer additional attempts than with standard messages, and by practical constraints on session length and participant fatigue. After each unsuccessful attempt, the editor resets to the original code. Success is acknowledged immediately, so as not to confound the results of the final evaluation task. In essence, this code fixing task addresses RQ1 by measuring repair effectiveness under each message condition and skill level.
3. **Error Message Evaluation:** Regardless of repair outcome in the previous task, participants are asked to rate the error message they saw using 5-point Likert scales (see Figure 3.7d). For this final task, we display the error message they had received in the previous task in addition to the actual statements to be rated, which will measure readability, cognitive load, and the authoritativeness of the error message (see Table 3.5 displaying all of the questions against which each error message was rated). To that end, readability is captured via four items, namely, length, jargon, sentence structure, and vocabulary. These items, both their wording and Likert scales, are adopted verbatim from previous work [16], which showed they correlate with readability and understanding. Cognitive load is probed along intrinsic, extraneous, and germane dimensions, as inspired by cognitive load theory (CLT) [58]. Finally, while there currently does not exist a validated instrument for measuring the authoritativeness of error messages, we capture this via a single item centered around the tone and empathetic quality of the error message. Overall, the error message evaluation task addresses RQ2 by testing whether LLM-enhanced PEMs are more readable, less demanding, and less authoritative in tone than Python standard messages.

MCQ score (correct)	Python years of experience	Assigned skill level category
≥ 6	Any	Expert
≤ 3	Any	Novice
$[4, 5]$	≥ 5	Expert
$[4, 5]$	< 5	Novice

Table 3.4: Logic for assigning participants to skill level categories based on MCQ score and years of Python experience (YoE).

Metric	Question	Scale (1-5)
Readability		
Length	"*Is the message expressed using more words than needed?*"	Succinct - Verbose
Jargon	"*Does the message contain jargon and technical terms?*"	Less - More
Sentence Structure	"*How clear is the sentence structure of the message?*"	Clear - Unclear
Vocabulary	"*How complex is the vocabulary used?*"	Simple - Complex
Cognitive load		
Intrinsic Load	"*This error message was inherently difficult to understand.*"	Disagree - Agree
Extraneous Load	"*The wording or formatting of this error message wasted mental effort.*"	Disagree - Agree
Germane Load	"*This error message helped me recognize the underlying error concept.*"	Disagree - Agree
Authoritativeness		
Tone (respectfulness)	"*How respectful (i.e., reader-centered) is the tone of the error message?*"	Disrespectful - Respectful

Table 3.5: Survey items for readability, cognitive load, and authoritativeness, with response scales.

4

Experimental Setup

This chapter describes the design of the main study conducted on Prolific to evaluate whether rephrased Python error messages, specifically *pragmatic* or *contingent* error message styles, improve participants' ability to correct buggy code compared to the *standard* interpreter messages (RQ1). Beyond repair effectiveness (i.e., fix rate, time-to-fix, number of attempts to fix), we assess participants' perceptions of *readability*, *cognitive load*, and *authoritativeness* of the error message they received (RQ2). The main crowdsourced study we launched consisted of three parts: (1) a Python MCQ skill assessment, (2) a single code-fixing task when being shown one error message style, and (3) an evaluation of the error message that was used in the previous task via Likert scales (see Table 3.5). Procedural details of these parts are described in Chapter 3, the research methodology. In this chapter we focus on formalizing the study design, by describing the independent variables in Section 4.1, dependent variables in Section 4.2, and possible confounding factors within Section 4.3. Lastly, in Section 4.4 we also report on more details surrounding our recruitment of participants via Prolific.

4.1. Independent Variables

This section specifies the factors that structure the study design and assignment. We control the *error message style* as the primary factor, assign the *code snippet and corresponding PEM* to control error diversity and difficulty, and measure *Python proficiency* to enable stratified balancing and subgroup analyses. Assignment uses constrained randomization to preserve global balance across styles and local balance within the determined code snippet group.

4.1.1. Error Message Style

Participants were pseudo-randomly assigned to one of three error message styles that were displayed alongside the offending code snippet:

- **Standard**: The unmodified Python interpreter message corresponding to the triggered error. This serves as the baseline against which rephrased variants are evaluated.
- **Pragmatic**: A concise, task-oriented rephrasing aimed at suggesting actionable next steps (e.g., what to check or change immediately) while avoiding extraneous detail. The goal is to lower the cognitive load for a fix without altering the underlying diagnostic content.
- **Contingent**: A scaffolded, stepwise rephrasing formulated using Toulmin's argumentation model to guide understanding and diagnosis (e.g., stating the claim, offering grounds, and prompting likely checks). This format completely changes the original error while making the reasoning path for a cause and fix explicit.

To reduce confounds, all rephrasings for a given snippet were authored from the same base interpreter message, and their length/structure were kept within pre-defined bands. Constrained randomization maintained *global* balance across styles and *local* balance within each snippet, and each participant saw exactly one error message style (i.e., no cross-over).

4.1.2. Code Snippet and Error Message Type

Each participant received exactly one of four curated Python snippets (which we encode as A, B, C, D). Every snippet contains a single targeted bug with a unique ground-truth fix and was accompanied by a hidden test suite to validate semantic correctness. Code snippet choice functions as a blocking factor to control for difficulty variability: balanced assignment ensured comparable counts per style within each snippet, thereby mitigating confounding from snippet-specific characteristics.

Naturally, the underlying *error message type*, i.e., the interpreter's error category raised by the snippet, such as `NameError`, always represents the error message that was triggered when running the assigned code snippet, and is the one which is rephrased into the pragmatic and contingent variants without changing its meaning or scope.

4.1.3. Python Proficiency Level

Participants first completed a Python skill assessment comprising (1) self-reported years of experience via a slider and (2) eight MCQs selected as best-performing in the pilot (see Section 3.1). The backend logic combined these measures to assign participants to proficiency strata (e.g., novice vs. expert), which were then used to:

1. **Support balanced assignment:** Stratified constrained randomization preserved style balance within snippet while maintaining similar skill distributions across conditions.
2. **Enable subgroup analyses:** Skill level was not manipulated, but explicitly assessed (e.g., style $\times$ skill interactions).

This approach limits bias from self-reporting measures alone by combining years of Python experience with objective (MCQ) indicators, while maintaining the grouping strictly at the message-style level.

4.2. Dependent Variables

This section outlines the study's dependent variables, which fall into two broad categories: (1) *behavioral outcomes*, which capture objective measures of participants' performance during the code-fixing task, and (2) *perceptual outcomes*, which reflect subjective ratings of the error message they received. Behavioral outcomes quantify the ability and efficiency with which participants corrected the targeted bug, while perceptual outcomes assess the clarity, cognitive demand, and tone of the error messages.

4.2.1. Behavioral Outcomes

Fix Rate: For each experimental group g (e.g., defined by message style $\times$ skill level, or message style $\times$ code snippet), we define *Fix Rate* as the proportion of participants who successfully repaired the snippet within at most three attempts, relative to all participants in that group. Formally:

$$\mathrm{FR}_g = \frac{\sum_{i \in g} \mathbf{1}(S_{i,1} = 1 \ \vee \ S_{i,2} = 1 \ \vee \ S_{i,3} = 1)}{\sum_{i \in g} \mathbf{1}(S_{i,1} = 0 \ \wedge \ S_{i,2} = 0 \ \wedge \ S_{i,3} = 0) + \sum_{i \in g} \mathbf{1}(S_{i,1} = 1 \ \vee \ S_{i,2} = 1 \ \vee \ S_{i,3} = 1)} \tag{4.1}$$

Note: FR_g = Fix Rate for group g, where $S_{i,k} \in \{0, 1\}$ encodes whether participant i's submission at attempt k was correct (1) or incorrect (0).

Time-to-Fix: For each participant i who achieved a successful fix, *Time-to-Fix* is the total elapsed time (in seconds) from message display to the successful attempt, summing the durations of all attempts leading up to and including the successful one:

$$\mathrm{TTF}_i = \sum_{k=1}^{K_i} t_{i,k} \tag{4.2}$$

Note: TTF_i = Time-to-Fix for participant i, where $t_{i,k}$ is the time taken for attempt k, and K_i is the index of the first successful attempt for participant i (with $1 \leq K_i \leq 3$). This metric is undefined for participants without a successful fix.

Fix@*k* Metrics: We further define and record the following metrics:

- **Fix@1:** Number of participants whose first attempt was successful at fixing the given buggy Python code snippet, i.e., $S_{i,1} = 1$.
- **Fix@2:** Number of participants whose second attempt was the first successful one at fixing the given buggy Python code snippet, i.e., $S_{i,1} = 0 \wedge\ S_{i,2} = 1$.
- **Fix@3:** Number of participants whose third attempt was the first successful one at fixing the given buggy Python code snippet, i.e., $S_{i,1} = 0 \wedge\ S_{i,2} = 0 \wedge\ S_{i,3} = 1$.

Note: $S_{i,k} \in \{0, 1\}$ encodes whether participant i's submission at attempt k was correct (1) or not (0).

4.2.2. Perceptual Outcomes

Following the code-fixing task, participants rated the error message they received on a 5-point Likert scale for three constructs:

- **Readability** (4 items): length, jargon, sentence structure, vocabulary.
- **Cognitive Load** (3 items): intrinsic, extraneous, germane load (per Cognitive Load Theory).
- **Authoritativeness** (1 item): perceived tone and empathetic quality, with higher scores indicating a less authoritative, more considerate tone.

Some scales were designed so that higher values indicate more desirable perceptions, while others were reverse-coded[1].

- **Readability items:** lower ratings reflect greater readability, therefore are better.
- **Cognitive load items:** lower ratings indicate reduced load and are therefore more desirable, except for germane load, where higher ratings are better.
- **Authoritativeness:** measured with a single item, where higher ratings represent a more reader-centered, and thus a better perceived error message.

4.3. Confounding Factors

We identified potential confounds and implemented our design to account for them:

- **Skill heterogeneity:** Individual ability can affect both fix rate and perceptions. We measured skill, namely Python years of experience and the number of correct multiple-choice questions, and used it for stratified balancing at assignment and for subgroup/adjusted analyses.
- **Snippet difficulty:** To mitigate difficulty-driven effects, we curated four single-bug snippets, piloted them for comparability, and blocked randomization by snippet. In our analysis, we sometimes treat the snippet choice as a factor.
- **Message length/content differences:** Pragmatic and contingent messages naturally differ in length/structure, yet this was an intentional introduced covariate.
- **Prior familiarity with specific errors:** Pre-existing familiarity may ease fixes. We use random assignment and snippet blocking to distribute such familiarity across conditions, thereby residual effects are partially mitigated by code snippet assignment logic.
- **Editor/task familiarity:** Participants interacted with the code editor interface during the survey's mandatory instructions step to reduce UI-driven cognitive load. The editor resets after attempt failures to avoid cumulative code drift.
- **External assistance:** As with any remote study, undisclosed external help cannot be fully excluded. We did instruct participants to work independently, with time-on-task, window/copy/paste events, and attempt patterns enabling us to detect implausibly rapid successes or other signs of malicious intent. In addition, copying and pasting text was disabled by default within the browser.

[1] We did not perform numerical reverse-coding, such as computing $x' = 6 - x$ when higher raw scores indicated worse outcomes. Instead, we kept the original coding and interpreted the direction of the scales when reporting results. Exact item wording and scales are provided in Table 3.5.

- **Order and fatigue effects:** Each participant fixes only one snippet under one message condition, then immediately rates that same message. This avoids cross-message carryover and ordering effects.
- **Randomization balance:** Constrained randomization ensured approximately equal allocation across message styles globally and within the code snippet subgroup, thereby allowing us to perform meaningful analyses and comparisons across groups.

4.4. Prolific Configuration

We recruited participants via Prolific, adhering to the platform's ethical and compensation guidelines. This section reports the recruitment configuration, allocation, and how we ensured high-quality data for our subsequent analysis. All experimental tasks were hosted on a dedicated virtual machine configured with 36 GB RAM, an NVIDIA GeForce RTX 2080 Ti GPU with 12 GB VRAM, and an Intel Xeon Gold 6226 CPU @ 2.70GHz. These specifications ensured efficient execution of code submissions within the embedded editor and provided sufficient capacity to deliver rephrased error messages to all users via Ollama[2].

4.4.1. Recruitment & Screening

Participants were required to meet several eligibility criteria configured via Prolific's built-in screeners. These included being at least 18 years old, fluent in English, having prior Python programming experience (as determined by Prolific's Python proficiency screener), and maintaining an approval rate above 90% on previous Prolific studies. We additionally required access to a desktop or laptop computer, as the task involved code editing in an embedded editor and was unsuitable for mobile devices. To avoid carryover effects, participants who had taken part in our earlier pilot study, conducted to identify the final four code snippets used in this experiment, were excluded.

4.4.2. Attention and Quality Monitoring

No explicit attention-check questions were included in the survey. Instead, data quality was safeguarded through several monitoring measures. We logged and analyzed browser focus events to detect prolonged tab switching or window minimization, prevented copy-paste actions via JavaScript, with in-browser notifications upon detection. We also recorded the `time_taken_ms` for each submission and examined timing patterns across all survey parts to flag implausibly fast completions or irregular submission intervals. Whenever any of these quality checks were triggered, participants' data was manually reviewed, and those failing these checks, such as seen to issue random-typing submissions, or repeatedly submitting the same code without making any changes to it, were rejected and removed from our final analysis.

4.4.3. Compensation and Duration

The study was initially estimated to take only 15-20 minutes. After a soft launch with 10 participants, the average completion time was observed to be approximately 24 minutes. As a consequence, we adjusted the Prolific listing to reflect this updated duration. After the entire recruitment ended, the observed mean completion time was 26 minutes, with a median of approximately 23 minutes. Participants were compensated at a rate of £9.00 per hour, 50% higher than the minimum payout on the platform, and consistent with Prolific's recommendation to ensure ethical pay and high-quality data collection.

4.4.4. Sample Size and Allocation

A total of 103 participants completed the study legitimately and passed all of the quality checks that we enforced (both manually and automatically). Allocation was balanced by design across error message styles and code snippet groups. Table 4.1 reports counts by *error message style* $\times$ *skill level*, and Table 4.2 reports counts by *code snippet ID* within each error message style. Once again, note that the counts in these tables reflect the final analyzed sample after excluding participants who were not eligible for our analysis, or had failed the data quality standards, as outlined in Sections 4.4.1 and 4.4.2.

[2] https://ollama.com/

	Standard	Pragmatic	Contingent	Total
Novice	13	13	12	38
Expert	22	22	21	63
Total	35	35	33	103

Table 4.1: Participant allocation by message style and skill level.

	A	B	C	D	Total
Standard	9	8	9	9	35
Pragmatic	9	9	8	9	35
Contingent	8	9	8	8	33
Total	26	26	25	26	103

Table 4.2: Participant allocation by snippet (A-D) within each message style.

5

Results

Having detailed our research methodology and experimental setup in the preceding chapters, we now present the results in an objective, sequential manner. We first report on the demographics of our Prolific participant user-base in Section 5.1. After this, we thoroughly cover the analysis and results of the Prolific main crowdsourced study in Section 5.2, which constitutes the bulk of our experimental evaluation and provides the primary evidence addressing our research questions.

5.1. Prolific Participants Demographics

For completeness, we report descriptive statistics on age, gender, and nationality for those Prolific participants whose demographic information was available, without collecting any personally identifying data, as shown in Figure 5.1. We do not conduct inferential analyses or subgroup comparisons on these demographics in this chapter, because several nationality categories and some age bands have limited counts, and such analyses are outside our research questions. Additional background measures such as years of Python experience and MCQ skill scores are summarized and, where appropriate, examined in the Exploratory Analysis in Section 5.2.3.

5.2. Objective and Subjective Outcomes

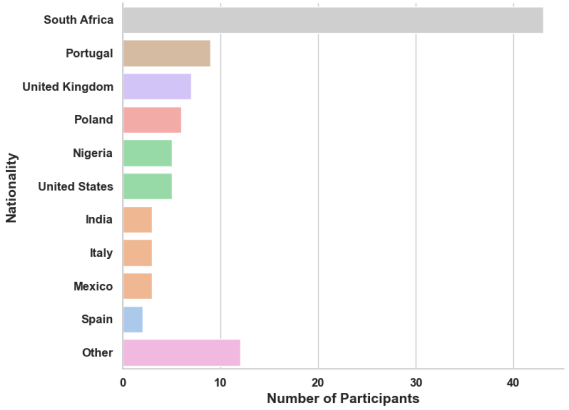
As previously explained in Section 4.4, the participants of our main study ($N = 103$) attempted to fix one of four buggy Python snippets (balanced across conditions) and were randomly assigned to one of three message styles: *Standard* (baseline), *Pragmatic*, or *Contingent*. We first report *objective performance*, such as fix rates, Fix@{1,2,3}, and time-to-fix in Section 5.2.1. We then report *subjective perceptions*, namely readability, perceived cognitive load, and perceived authoritativeness in Section 5.2.2. Finally, we also perform *exploratory analyses* that investigate additional potentially unexplored patterns in Section 5.2.3, in addition to the main results.

As a reminder for interpreting the results in the following sections, participants were classified according to our assignment logic, which combined their performance on the skill-assessment survey with their self-reported Python YoE. Out of the 103 participants, 65 were categorized as experts and 38 as novices. The distribution of participants across skill levels and snippets is provided in Table 4.1.

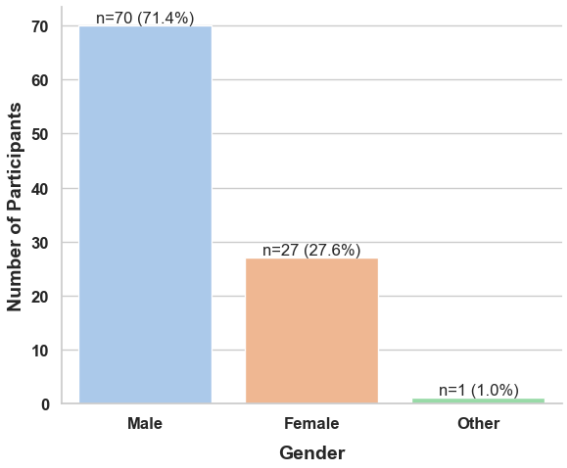
5.2.1. Fix Rate, Fix@k, and Time-to-Fix

For measuring participants' objective performance on the code fix task, across different subgroups we record the following metrics:

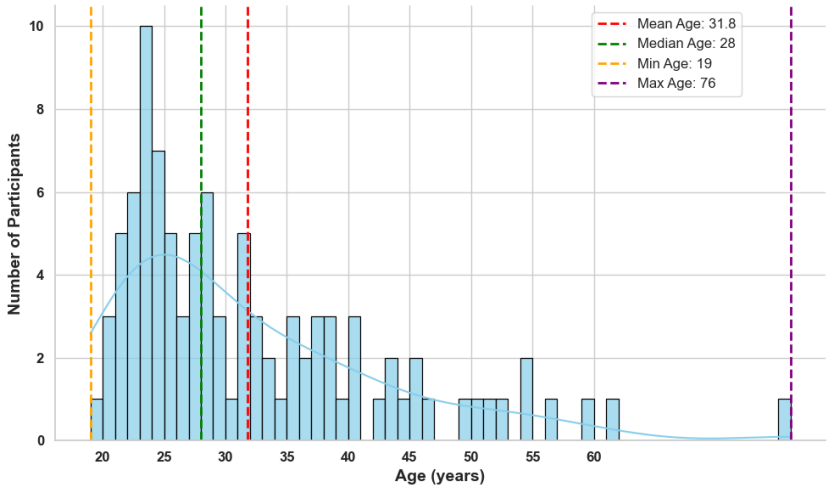
- **Fix rate**: proportion of participants whose final submission is fully correct.
- **Fix@1/2/3**: correctness achieved within the first, second, or third submission attempt.
- **Time-to-fix**: total seconds from code fix task start until the first correct submission.



(a) Participants' nationality distribution.



(b) Participants' gender distribution.



(c) Participants' age distribution.

Figure 5.1: Descriptive statistics for Prolific participants' demographics.

Fix Rate: Table 5.1 reports overall fix rates by snippet across all participants, and Table 5.2 breaks them down by message style and skill level. Fix rate, and therefore difficulty, varies significantly across snippets, with snippet A showing the lowest success rate. This outcome is expected because the standard interpreter output for snippet A fails to reveal the true root cause, and actually points to the non-offending line[1].

Looking more in-depth, Tables 5.3 and 5.4 report fix rates by message style and snippet ID for novices and experts. For snippet A, LLM-enhanced PEMs, which do correctly identify the offending line and root cause, generally increase fix rates, with the pragmatic style among novices as the main exception. However, across snippets, style effects vary widely, so we refrain from making any conclusions.

	A	B	C	D
Fix Rate	53% ($n = 26$)	84% ($n = 26$)	64% ($n = 25$)	69% ($n = 26$)

Table 5.1: Fix rate (%) by snippet ID, across all participants; n = the total number of participants assigned the given snippet ID.

[1]Interested readers can refer to Listing B.1 and Figure B.1 in Appendix B, which present the code and the corresponding standard error message for the least solved buggy code snippet in our study (when being shown the standard interpreter output).

	Standard	Pragmatic	Contingent
Novice ($n = 38$)	46%	54%	42%
Expert ($n = 65$)	68%	82%	90%

Table 5.2: Fix rate (%) by message style and skill group (all snippets pooled).

	A	B	C	D
Standard	33% ($n = 3$)	67% ($n = 3$)	33% ($n = 3$)	50% ($n = 4$)
Pragmatic	0% ($n = 3$)	100% ($n = 4$)	67% ($n = 3$)	33% ($n = 3$)
Contingent	67% ($n = 3$)	67% ($n = 3$)	0% ($n = 3$)	33% ($n = 3$)

Table 5.3: Fix rate (%) by message style and snippet ID for novice participants. The number n, in parentheses, represents the total number of participants assigned the given combination of snippet ID and error message style.

Fix@k: We initially introduced the Fix@k metric to quantify how many attempts participants needed to repair each buggy snippet. Our aim was to test whether error message style influenced repeated submissions across the full sample and within novice and expert subgroups. In practice, results were situated at the extremes. Most participants either fixed the error on the first try, or did not succeed at all within the three allowed attempts. Because Fix@k showed limited variation (Figure 5.2), we could not conduct meaningful further analysis, so Fix@k remains largely unexplored in this study. A visual reading of the figure nevertheless suggests that pragmatic and contingent messages may yield higher fix rates, in general, than the baseline, since they outperform standard at $k = 1$. For contingent messages in particular, outcomes often cluster at either "fixed on the first attempt" or "not at all within three attempts". Therefore, interpreting low Fix@2 and Fix@3 counts requires care, since they can result from either strong first-attempt performance or a large tail of unresolved cases. Future studies should relax the bound on k (within reason) to better visualize the average number of attempts per error-message style until successful code repair.

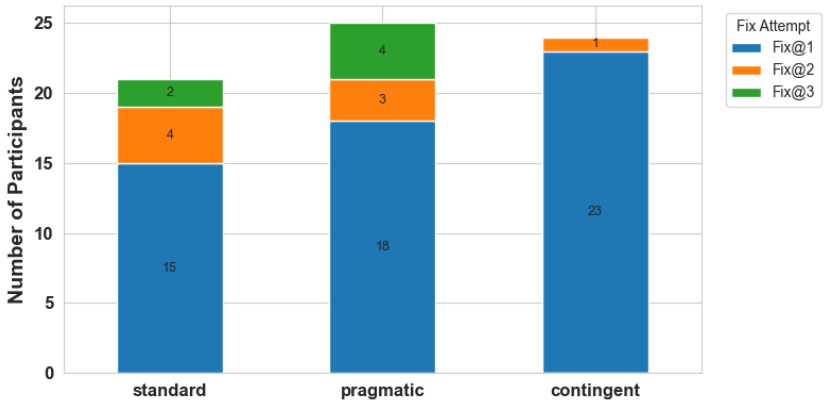


Figure 5.2: Fix@k, for $k \in 1, 2, 3$, when varying error message style across all snippets and skill levels.

Time-to-Fix: As for the time-to-fix metric, Figure 5.3 shows that, averaged across skill levels, participants fixed errors fastest when shown pragmatic messages. Contingent messages also reduced time-to-fix relative to the baseline, but less than pragmatic, which is unsurprising given their greater verbosity and the extra reading time required. However, when split by skill level (see Figure 5.4), both LLM-enhanced PEMs tend to speed fixes versus standard. With few novice successes (standard n=6, pragmatic n=7, contingent n=5), statistical tests found no significant effects, though the current pattern seems to favor LLM explanations. As for experts, results remain inconclusive.

	A	B	C	D
Standard	33% ($n = 6$)	80% ($n = 5$)	67% ($n = 6$)	100% ($n = 5$)
Pragmatic	83% ($n = 6$)	100% ($n = 5$)	80% ($n = 5$)	67% ($n = 6$)
Contingent	80% ($n = 5$)	83% ($n = 6$)	100% ($n = 5$)	100% ($n = 5$)

Table 5.4: Fix rate (%) by message style and snippet ID for expert participants. The number n, in parentheses, represents the total number of participants assigned the given combination of snippet ID and error message style.

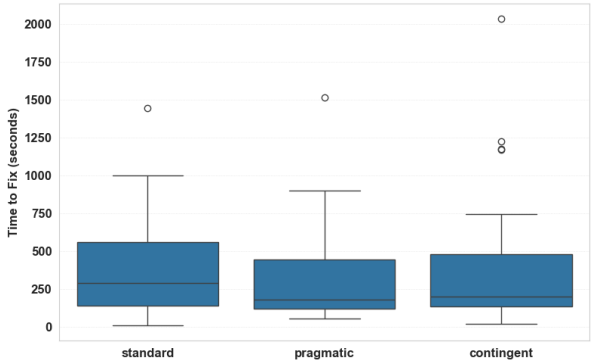


Figure 5.3: Box plot for time-to-fix (seconds) by message style. Time-to-fix sums all attempts until the first success. Standard ($M = 415.69, SD = 377.75$); Pragmatic ($M = 324.56, SD = 333.12$); Contingent ($M = 423.85, SD = 501.47$).

5.2.2. Subjective Evaluations

Figure 5.5 shows mean ratings by error message style and skill level. For **novices**, Holm-adjusted Dunn tests indicate lower *extraneous load* for Pragmatic vs. Standard ($p = 0.039$) and also for Contingent vs. Standard ($p = 0.054$). The results also report lower *authoritativeness* for Pragmatic vs. Standard ($p = 0.011$), while *vocabulary* also shows a trend favoring Pragmatic vs. Standard ($p = 0.087$), meaning Pragmatic messages are generally seen as using simpler vocabulary.

For **experts**, Pragmatic vs. Standard is associated with lower *intrinsic* ($p = 0.0049$) and higher *germane* load ($p = 0.0043$), and both Pragmatic and Contingent are rated less *authoritative* than the Standard ones ($p = 0.0097$ in both contrasts). Experts also perceive messages as more verbose for Contingent vs. Standard ($p = 0.035$), with a trend toward simpler *vocabulary* for Pragmatic vs. Standard ($p = 0.088$). Overall, adaptive style, especially Pragmatic, are consistently rated by experts as lighter in cognitive load, less authoritative, and more useful than the Standard baseline. Contingent error messages are also, on average, rated better by experts across almost all metrics, yet their increased verbosity seems to impact their overall usefulness.

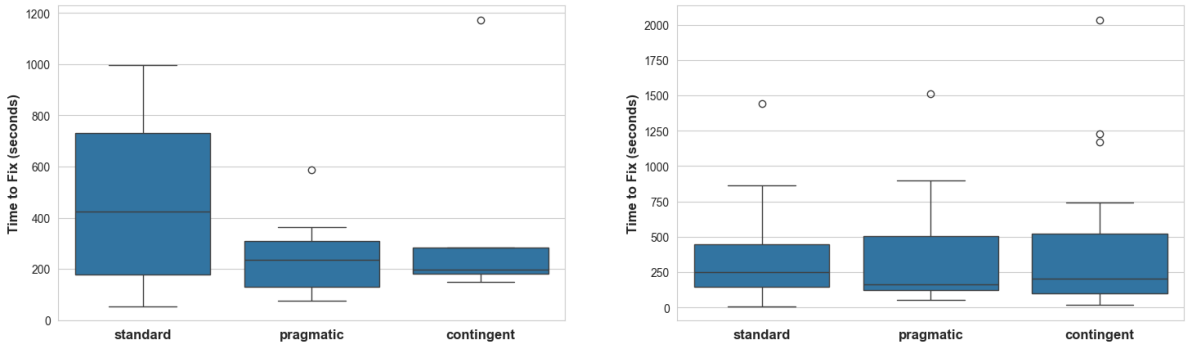
Figure 5.6 reports mean ratings for the full sample, aggregated only by message style. Similarly, we also perform Kruskal-Wallis tests followed by Holm-adjusted Dunn comparisons where necessary, and the results indicate several pairwise differences.

In terms of *readability*, differences appear for *sentence structure* between Pragmatic and Standard ($p = 0.0218$), for *vocabulary* between Pragmatic and Standard ($p = 0.0063$) and Pragmatic vs. Contingent ($p = 0.0423$), and a trend for *length* between Standard and Contingent ($p = 0.0949$).

For *cognitive load*, we observe differences in *intrinsic load* for Pragmatic vs. Standard ($p = 0.0017$) and Contingent vs. Standard ($p = 0.0431$). In *extraneous load* we notice differences only for Pragmatic vs. Standard ($p = 0.0064$). We also reported differences in *germane load*, both for Pragmatic vs. Standard ($p = 0.0013$) and Contingent vs. Standard ($p = 0.0488$).

Finally, in terms of *authoritativeness*, both Pragmatic and Contingent differ from the Standard error message style ($p = 0.0001$ and $p = 0.0032$, respectively).

Overall, the most frequent contrasts involve Pragmatic (and, in several cases, Contingent) relative to the Standard, consistent with adaptive styles differing from the baseline interpreter output on readability, perceived load, and tone.



(a) Time-to-fix box plot for novices, by message style. Standard ($M = 471.58, SD = 375.0$); Pragmatic ($M = 253.71, SD = 178.52$); Contingent ($M = 396.41, SD = 437.06$).

(b) Time-to-fix box plot for experts, by message style. Standard ($M = 393.34, SD = 389.57$); Pragmatic ($M = 352.13, SD = 377.56$); Contingent ($M = 431.08, SD = 527.85$).

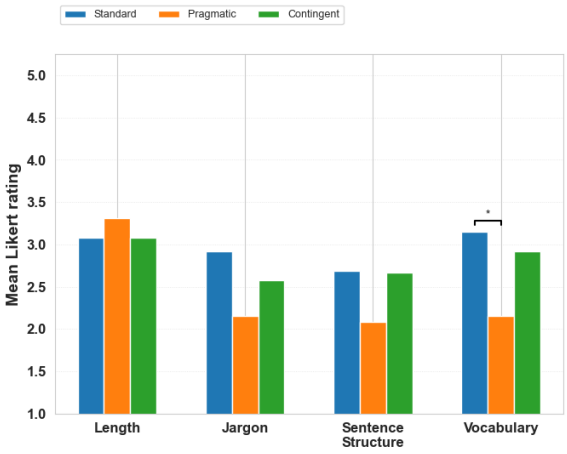
Figure 5.4: Box plots for time-to-fix (seconds) by message style and skill level; left plot for novices, and right plot for experts.

For the previously mentioned statistical analysis, we use the Kruskal-Wallis test with Dunn's pairwise comparisons (Holm-adjusted) because our outcomes are Likert ratings (i.e., ordinal, bounded), and after performing validation checks we saw that the data was not normally distributed. Kruskal-Wallis is an alternative to ANOVA that handles such data well and tolerates unequal group sizes. When the omnibus test indicates a difference, Dunn's test tells us which styles differ. Holm's adjustment controls false positives across the three pairwise checks per metric while being less conservative (and more powerful) than an alternative like Bonferroni. Overall, this is a data-appropriate choice for our setting.

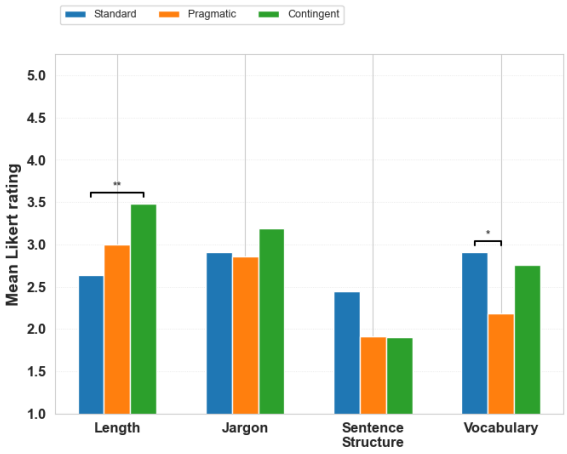
5.2.3. Exploratory Analysis

In addition to the previous results, we also performed additional exploratory analysis to see whether we could find uncovered patterns. For instance, Figure 5.7 depicts statistics about the multiple-choice questions that compose the Python skill assessment in the first part of the Prolific study, where we are reporting mean completion time and question correctness percentage. On average, we noticed that correct answers tend to require more time on the respective question, indicating that our questions were mostly answered thoughtfully and legitimately, rather than at random.

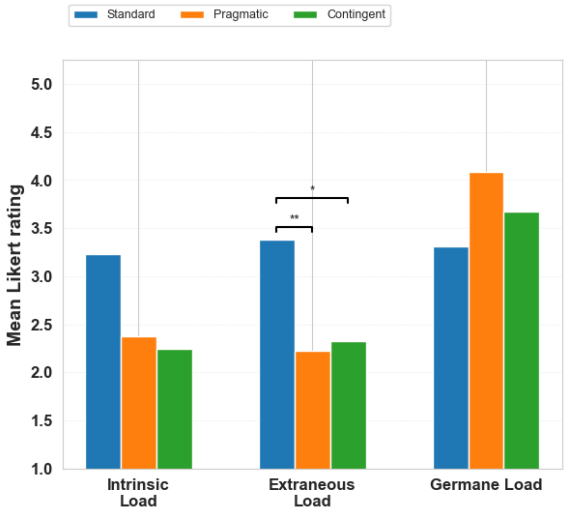
Another aspect we wanted to verify was whether the piloted Python skill assessment survey reliably reflects true proficiency and predicts debugging performance in the main Prolific study. To that end, we tested whether performance on the survey, measured as the number of MCQs answered correctly, relates to fixing the assigned code snippet. A point-biserial correlation showed a moderate positive association, $r_{pb} = 0.368$ with $p = 0.0001$, indicating that higher MCQ scores correspond to a greater likelihood of producing a correct fix.



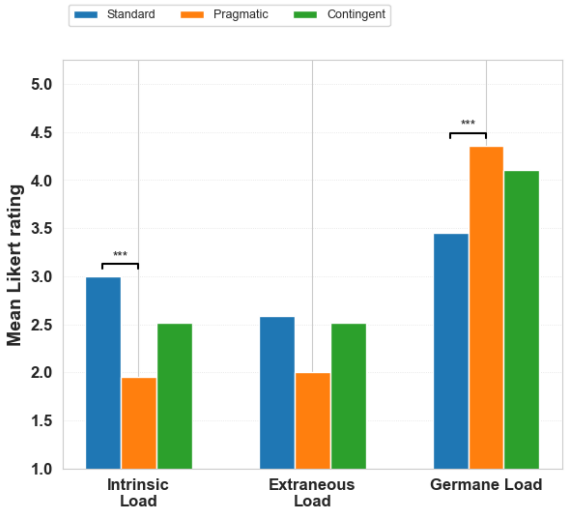
(a) Readability metrics (Novices).



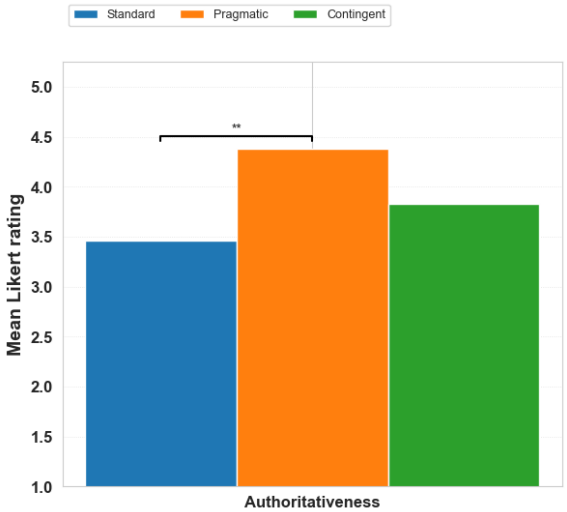
(b) Readability metrics (Experts).



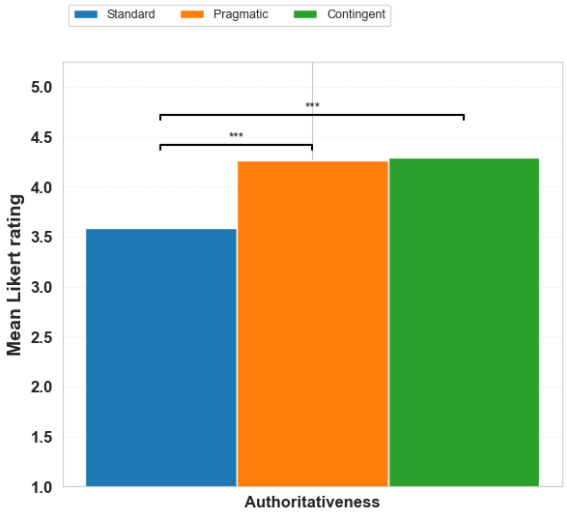
(c) Cognitive load metrics (Novices).



(d) Cognitive load metrics (Experts).



(e) Authoritativeness metric (Novices).



(f) Authoritativeness metric (Experts).

Figure 5.5: Mean ratings for readability, cognitive load, and authoritativeness by message style and skill group. Brackets with asterisks indicate pairwise differences from Dunn's post hoc tests (Holm-adjusted) following Kruskal-Wallis: * ($p < 0.1$), ** ($p < 0.05$), and *** ($p < 0.01$).

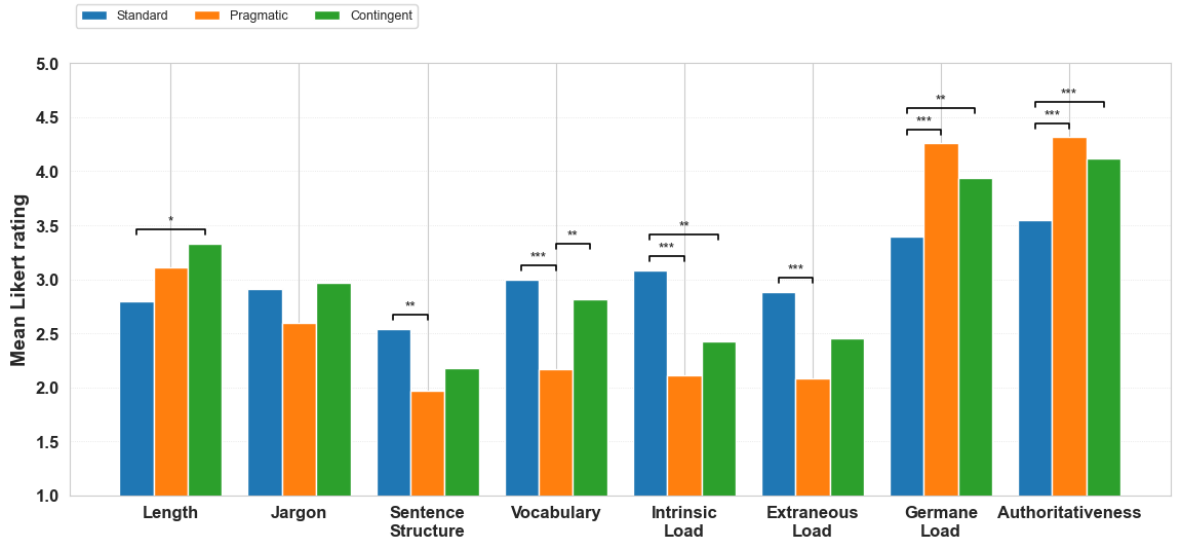
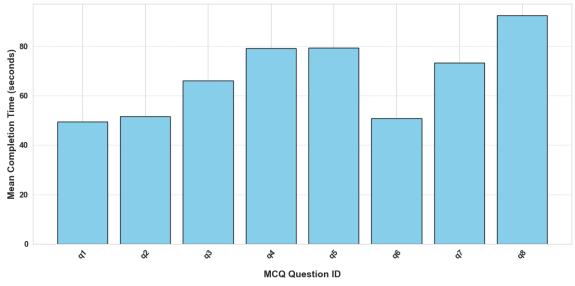
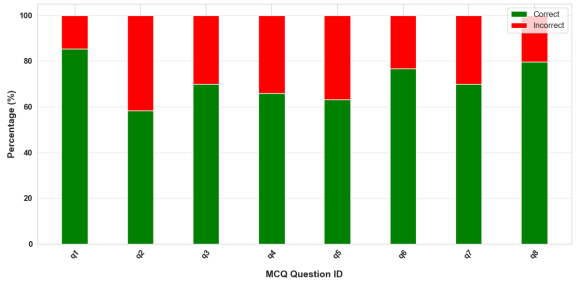


Figure 5.6: Mean ratings for readability, cognitive load, and authoritativeness by message style. Brackets with asterisks indicate pairwise differences from Dunn's post hoc tests (Holm-adjusted) following Kruskal-Wallis: * ($p < 0.1$), ** ($p < 0.05$), and *** ($p < 0.01$).



(a) Mean completion/answering time in seconds for each MCQ in the Prolific study.



(b) Percentage of correct and incorrect answers for each MCQ in the Prolific study.

Figure 5.7: Statistics, such as mean completion time and correctness percentage pertaining to the MCQs in the Prolific study.

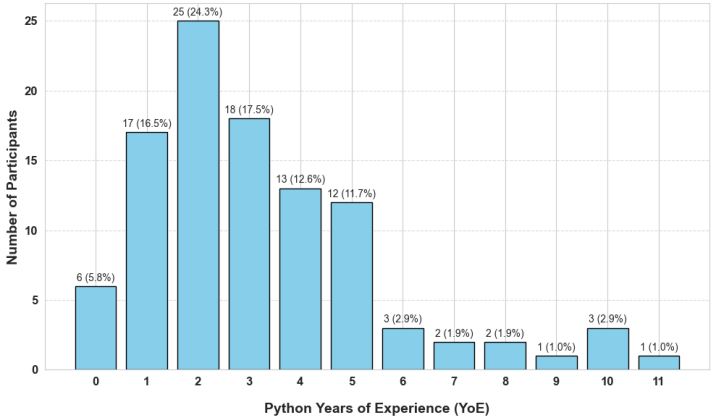


Figure 5.8: Prolific participants' self-reported Python YoE distribution.

6 Discussion

This chapter provides an in-depth discussion of our results, linking them back to the research questions posed in the introduction and considering their implications (Section 6.1). We then take a retrospective view of our work, outlining the limitations of our approach and suggesting potential directions for future work (Section 6.2).

6.1. Implications

The results of our study carry implications for both how programmers respond to error messages in practice and how they perceive their clarity and usefulness. We discuss these implications in relation to RQ1 (objective effectiveness) and RQ2 (perceived qualities).

6.1.1. RQ1: Effectiveness of Skill-Adaptive Programming Error Messages

The results from Section 5.2.1 show that our LLM-powered, skill-adaptive error messages reduced debugging time and improved fix rates for both novices and experts, though these improvements were not statistically significant compared to baseline interpreter messages. While this may appear disappointing, given the relatively small sample size which limits statistical power, we believe the trends are promising enough to warrant further study with larger populations. It is also important to note that not all code snippets used in the study were equally difficult, and error message effectiveness varied by error message style and participant skill level.

Looking at Fix@k metrics, which were meant to measure attempts and error message style effects, results generally indicated that participants either fixed the given code on the first try or not at all within three attempts, leaving too little variation for meaningful analysis. Interpreting Fix@1 alone may suggest that contingent messages are most effective (when looking at the entire population, and not when grouping by skill level), yet the sparse Fix@2 and Fix@3 counts could also imply that they are potentially confusing, with participants either understanding them from the beginning or not at all. Future work should revisit Fix@k, and allow a larger k in related experiments to better capture attempt effects.

Finally, we were particularly surprised to find that contingent error messages did not consistently improve the fix rate for novices when comparing against both baseline and pragmatic, despite being designed to provide more scaffolding and context through an argumentation-based structure. A plausible explanation is that the increased verbosity of contingent messages led novices to spend more time reading and interpreting them rather than applying correct fixes, offsetting their potential benefits. Still, fix rate and time-to-fix are only two metrics, and contingent messages may yet prove valuable in other respects. Interestingly, contingent messages did significantly improve fix rates for experts, even compared to pragmatic ones, suggesting that detailed, structured feedback can actually benefit experienced programmers, despite increased verbosity.

6.1.2. RQ2: Perceived Qualities of Skill-Adaptive Programming Error Messages

Turning to subjective perceptions, as reported in Section 5.2.2, participants of all skill levels tended to rate LLM-enhanced PEMs, pragmatic and contingent, more favorably than the standard Python interpreter ones. They found them clearer, more helpful for understanding the underlying issue, lower in intrinsic cognitive load, and less authoritative. Experts strongly preferred pragmatic messages, which they judged better across most subjective dimensions (with significant results for vocabulary, intrinsic load, germane load, and authoritativeness). This aligns with our hypothesis that concise, to-the-point feedback is more suitable for experienced users than scaffolded information. The only anomaly that we encountered was that experts perceived pragmatic messages as more verbose than standard ones, even though they were objectively shorter. We attribute this either to differences in how participants interpreted "verbosity" (textual vs. structural conciseness) or to possible confusion about the survey scales and their actual meaning (1 = very succinct, 5 = very verbose).

For novices, subjective ratings followed a similar pattern. Pragmatic messages were generally preferred over contingent and baseline ones, with significant results for vocabulary, extraneous load, and authoritativeness. Contrary to our hypothesis, novices did not perceive contingent PEMs as the most useful. We suggest that highly detailed and scaffolded feedback may actually overwhelm learners still grappling with the basics, leading to cognitive overload rather than support. Another possibility is that some participants of our "novice" group, as categorized by our skill assessment survey in the first part of the Prolific study, were in fact more intermediate-level programmers and not "true" novices. While we initially considered finer-grained skill categorization, we opted for a strict novice/expert split to preserve clarity in our factorial design.

In summary, even with a modest sample size of 103 participants, our study indicates that LLM-powered, skill-adaptive error messages can support both novices and experts. Pragmatic messages appear most consistently effective, while contingent messages surprisingly offer benefits mainly for experts. These findings align with broader trends toward personalized and adaptive developer support using LLMs. At the same time, they highlight the need for caution: such tools are not a "silver bullet" and should augment, rather than replace, existing debugging practices and human instruction. In educational contexts, over-reliance on LLM feedback, particularly if explanations are overly confident but incorrect, risks strengthening misconceptions, false confidence, and inhibiting learning [26]. This highlights the importance of balancing adaptive AI support with opportunities for learners to develop independent debugging and problem-solving skills [7].

6.2. Looking Back and Ahead

Our findings show significant promise, but they also reveal some limitations in how much our methods can currently be generalized. In this section, we reflect on those limitations and indicate concrete directions for extending and testing our approach in more realistic and varied settings.

6.2.1. Limitations

Our study has several limitations that need to be considered when interpreting our findings. First, although we aimed to develop a more reliable evaluation of programming skills than self-reported measures, the skill-assessment survey we created via our first pilot study, and which we used in our main study, was not empirically validated at a large scale before launch. Additionally, in the Prolific study, participants were grouped by skill using this survey, together with self-reported Python years of experience, using the deterministic grouping logic described in Table 3.4. This rule was a pragmatic choice given our study constraints, rather than a validated scoring model, and we did not compare it against alternatives or correlate it with external criteria such as course grades, other exams performance or any other sort of user-provided certifications. Given the heterogeneity of programmer backgrounds, no single measure could possibly perfectly capture skill level. Nevertheless, stronger validation of this skill-assessment survey would improve confidence in the subgroup analyses of Chapter 5.

A second limitation concerns task coverage for the code fix task in the Prolific main study and its ecological validity. Even after a second pilot reduced the set of eight initial synthetic candidate snippets to the four we used, our buggy Python code snippets still captured only a limited range of error contexts. Real-world failures span diverse categories (e.g., `NameError`, `SyntaxError`, `IndentationError`, `TypeError` and much more) and typically arise in code with far greater variability than our concise ($\leq$ 60 lines of

code) snippets designed to be feasible for Prolific participants. Effects may therefore differ for larger programs, multi-file projects, or complex build and tooling pipelines. Generalization to such contexts may require larger models with longer context windows or more sophisticated prompting and retrieval strategies.

Third, in selecting our model and prompting strategy, we prioritized practicality rather than breadth. We used `llama3.1:8b` at temperature 0 with zero-shot prompting to support low-latency, concurrent inference on a single VM for $\sim$100 participants and to ensure stable, "deterministic" outputs, which are not overly biased on prior exemplars. Although our choices were pragmatic and consistent with prior work and benchmark results for the selected model, they also limited the scope of exploration. A combination of different settings, such as larger models, or different prompting strategies (e.g., one-/few-shot, chain-of-thought, or other types of unexplored prompts) might have produced different outcomes. Likewise, we relied on single templates for pragmatic and contingent styles without adapting them to individual error types, which may have left some potential gains unexplored.

Finally, scope and power also constrain generalizability, especially since all experiments were conducted in Python. Although several design ideas (e.g., prompt structure and information provided to the model) may transfer, languages with different error types and compiler/runtime feedback (e.g., Rust, Haskell, Java) may require some slight adaptation. Our post-quality check sample (N=103) is relatively modest, which limits statistical power. While some of our interventions did yield statistically significant effects, the small sample size nonetheless constrains the strength and generalizability of these findings. Additionally, a few subjective items within our Prolific main study (e.g., perceived verbosity) may have been interpreted differently despite careful wording and anchors, suggesting room to refine instrument design and reverse-coded clarity.

6.2.2. Future Work

The aforementioned limitations suggest several ideas worth investigating in follow-up studies. A first priority is validation in an authentic educational setting, where true novice programmers would be evaluating our skill-adaptive error messages. We propose a controlled deployment in an introductory programming course over 1-3 weeks, randomizing at the participant or assignment level to compare baseline, pragmatic, and contingent messages (or an adaptive selector). Beyond immediate outcomes (fix rate, time-to-fix), such a deployment should track learning outcomes (e.g., assignment/exam performance, delayed post-tests for retention checks) and correlate them with multiple skill measures, such as MCQs, instructor rubrics, prior GPA, etc.

A second avenue for future work is evaluation at scale when triggering errors for large, real-world codebases. An IDE plugin (e.g., VS Code or PyCharm) could surface our LLM-enhanced PEMs alongside standard interpreter output via an opt-in toggle or randomized exposure, thus enabling between or within-subject comparisons at scale. Fine-grained telemetry, such as time-to-fix, edit distance to fix, error recurrence, hint or toggle requests, should be collected under strict privacy safeguards. Complementary offline benchmarks (e.g., log replay from IDE failures or historically common issues) would provide controlled comparability across teams and repositories.

Moving beyond static styles, a third direction is genuine skill adaptivity. Rather than pre-assigning novices and experts to fixed groups, future systems could adapt dynamically by estimating transient user skill state from behavioral signals (i.e., recent success rate, fix latency, error recurrence patterns, scope of edits, number of failed runs). Methods such as contextual multi-armed bandits or reinforcement learning could select message style, granularity, and tone in real time. Exposing a "skill/state" control variable to the LLM would allow direct tests of whether the model reliably conditions its explanations. Note, however, that this would represent a non-trivial extension to our framework.

Fourth, there is room to explore richer ways of testing and personalizing our approach. Future studies could try different model sizes, temperatures, decoding methods, or prompt templates, perhaps tied to specific error types, to see how they affect clarity, speed, and accuracy. Error messages could also be made more helpful through retrieval-augmented explanations (e.g., linking to examples or documentation) and by letting users adjust the level of detail. Also, developing a specialized, fine-tuned model for explaining diverse error messages in an educational or simple natural language style has the potential to further improve the debugging and learning experience. In addition, fairness and accessibility

need to be considered, including readability for non-native speakers, support for dyslexic readers, and attention to color and contrast.

Fifth, broader generalization is needed. Replications in other languages and environments (e.g., compiled languages, or even for build tools, such as GitHub Actions, Azure DevOps, etc.) and extensions from medium-sized code snippets to project-scale interactions would test ecological robustness. Longer term outcomes, such as debugging self-efficacy, the formation of accurate mental models, and the risk of over-reliance on LLM explanations should be evaluated to ensure that short term gains are not achieved at the cost of genuine understanding. Prior work on the benefits and harms of generative AI for novice programmers found that GenAI can amplify existing metacognitive difficulties among struggling students [49], which reinforces the need to monitor these effects.

Finally, both measurement and openness deserve greater attention. Subjective survey instruments could be improved with clearer examples of concepts such as “verbosity”, explicit checks for scale orientation, and reporting of reliability (e.g., Cronbach’s α[1]). Subjective measures of cognitive load could also be complemented by lightweight behavioral proxies, such as pause times before edits or navigation events, if using an IDE to assess performance. Moreover, continuing to publicly release prompts templates, alongside a synthetic yet realistic benchmark of errors spanning multiple types and code sizes, would strengthen reproducibility and enable cumulative comparison across future studies.

In summary, this work provides initial evidence that skill-aware programming error messages can be beneficial, though their effectiveness cannot be proven to be consistent across all users and contexts, *yet*. The critical next step is to test them in larger, more ecologically valid settings, paired with dynamic skill estimation and adaptive customization, to move toward genuinely effective, personalized debugging support.

[1] Cronbach’s alpha is an index of internal consistency that shows how closely a set of items is related, essentially reflecting whether they measure the same construct [60].

7

Conclusion

This thesis proposed a framework for designing LLM-powered, skill-aware Python programming error messages aimed at improving readability, reducing cognitive load, and adopting a more reader-friendly tone compared to the standard interpreter output. While prior work has explored ways to improve programming error messages across languages, many approaches are still inherently rigid and struggle to accommodate differences in expertise and learning preferences. Recent advances in large-language models open up intriguing opportunities for more tailored feedback.

Motivated by the need for personalized support at the point of failure, we developed and evaluated two styles of LLM-powered error messages with both novice and expert participants. *Pragmatic* messages were designed to be concise and action-oriented, whereas *contingent* messages provided scaffolded, actionable guidance structured by a clear argumentation model. To operationalize skill, we designed and piloted a brief Python assessment focused on debugging and understanding interpreter errors. The scores from this assessment determined the novice/expert grouping used in our evaluation.

In our study, pragmatic messages were, on average, more effective than both the baseline and contingent variants, helping programmers fix issues with less effort and, often, in less time. Contingent messages also outperformed the baseline but generally trailed behind the pragmatic style, especially when considering time-to-fix metrics. Objectively, improvements in fix rate and time were not statistically significant. However, subjective ratings showed clear, statistically significant advantages for the LLM-generated messages, with pragmatic, to-the-point guidance proving broadly effective. Participants in our study, novices and experts alike, judged pragmatic messages to be more readable, lower in intrinsic cognitive load, more helpful for understanding the underlying issue, and less authoritative in tone. By contrast, contingent messages, though informative, sometimes increased the perceived intrinsic cognitive load. This was especially true for novices, suggesting that verbosity, even when correct and relevant to the context, can introduce cognitive overhead contrary to our initial hypothesis.

This work is an initial promising step covering personalized error message design, with numerous opportunities for extension. Future research should (1) evaluate the approach at a larger scale in authentic educational and industrial settings, (2) generalize beyond Python to languages with different error surfaces, and (3) move toward true adaptivity, where the message style and level of detail adjust dynamically to signals of user skill and task state. Pursuing these directions will bring us closer to genuinely personalized, effective debugging support: error messages which adapt to you and your experience as you grow and learn more.

References

[1] Abdulaziz Alaboudi and Thomas D. LaToza. “What constitutes debugging? An exploratory study of debugging episodes”. In: *Empirical Softw. Engg.* 28.5 (Sept. 2023). ISSN: 1382-3256. DOI: 10.1007/s10664-023-10352-5. URL: https://dl.acm.org/doi/10.1007/s10664-023-10352-5.

[2] Anthropic. *Building effective agents*. Anthropic Engineering Blog. https://www.anthropic.com/engineering/building-effective-agents. Dec. 2024.

[3] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Anthropic model card. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf. Mar. 2024.

[4] Patricia Armstrong. “Bloom’s taxonomy”. In: *Vanderbilt University Center for Teaching* 12.05 (2010), p. 2023.

[5] Jacob Austin et al. *Program Synthesis with Large Language Models*. 2021. arXiv: 2108.07732 [cs.PL]. URL: https://arxiv.org/abs/2108.07732.

[6] Titus Barik et al. “How should compilers explain problems to developers?” In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 633–643. ISBN: 9781450355735. DOI: 10.1145/3236024.3236040. URL: https://dl.acm.org/doi/10.1145/3236024.3236040.

[7] Elisabeth Bauer et al. “Looking Beyond the Hype: Understanding the Effects of AI on Learning”. In: *Educational Psychology Review* 37 (Apr. 2025), pp. 1–27. DOI: 10.1007/s10648-025-10020-8.

[8] Brett A Becker et al. “Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages”. In: *SIGCSE ’18*. New York, NY, USA: Association for Computing Machinery, Feb. 2018, pp. 634–639. DOI: 10.1145/3159450.3159453. (Visited on 12/13/2023).

[9] Brett A. Becker et al. “Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’19. Aberdeen, Scotland Uk: Association for Computing Machinery, 2019, pp. 177–210. ISBN: 9781450375672. DOI: 10.1145/3344429.3372508. URL: https://dl.acm.org/doi/10.1145/3344429.3372508.

[10] Peter Belcak et al. *Small Language Models are the Future of Agentic AI*. 2025. arXiv: 2506.02153 [cs.AI]. URL: https://arxiv.org/abs/2506.02153.

[11] Patricia Benner. “Using the Dreyfus model of skill acquisition to describe and interpret skill acquisition and clinical judgment in nursing practice and education”. In: *Bulletin of science, technology & society* 24.3 (2004), pp. 188–199.

[12] Tessa Charles and Carl Gwilliam. “The Effect of Automated Error Message Feedback on Undergraduate Physics Students Learning Python: Reducing Anxiety and Building Confidence”. In: *Journal for STEM Education Research* 6.2 (Aug. 2023), pp. 326–357. ISSN: 2520-8713. DOI: 10.1007/s41979-022-00084-4.

[13] Anwoy Chatterjee et al. *POSIX: A Prompt Sensitivity Index For Large Language Models*. 2024. arXiv: 2410.02185 [cs.CL]. URL: https://arxiv.org/abs/2410.02185.

[14] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[15] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[16] Paul Denny et al. “On Designing Programming Error Messages for Novices: Readability and its Constituent Factors”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445696. URL: https://dl.acm.org/doi/10.1145/3411764.3445696.

[17] Robert Dyer and Jigyasa Chauhan. “An exploratory study on the predominant programming paradigms in Python code”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 684–695. ISBN: 9781450394130. DOI: 10.1145/3540250.3549158. URL: https://dl.acm.org/doi/10.1145/3540250.3549158.

[18] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. “Common logic errors made by novice programmers”. In: *Proceedings of the 20th Australasian Computing Education Conference*. ACE ’18. Brisbane, Queensland, Australia: Association for Computing Machinery, 2018, pp. 83–89. ISBN: 9781450363402. DOI: 10.1145/3160489.3160493. URL: https://dl.acm.org/doi/10.1145/3160489.3160493.

[19] Ujwal Gadiraju et al. “Using Worker Self-Assessments for Competence-Based Pre-Selection in Crowdsourcing Microtasks”. In: *ACM Trans. Comput.-Hum. Interact.* 24.4 (Aug. 2017). ISSN: 1073-0516. DOI: 10.1145/3119930. URL: https://dl.acm.org/doi/10.1145/3119930.

[20] Joshua Garcia et al. “A comprehensive study of autonomous vehicle bugs”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 385–396. ISBN: 9781450371216. DOI: 10.1145/3377811.3380397. URL: https://dl.acm.org/doi/10.1145/3377811.3380397.

[21] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[22] Daya Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE]. URL: https://arxiv.org/abs/2401.14196.

[23] Muhammad Usman Hadi et al. *Large Language Models: A Comprehensive Survey of its Applications, Challenges, Limitations, and Future Prospects*. July 2023. DOI: 10.36227/techrxiv.23589741.v1.

[24] Mark Halper. “How Software Bugs led to “One of the Greatest Miscarriages of Justice” in British History”. In: *Commun. ACM* 68.3 (Feb. 2025), pp. 12–14. ISSN: 0001-0782. DOI: 10.1145/3703779. URL: https://dl.acm.org/doi/10.1145/3703779.

[25] Björn Hartmann et al. “What would other programmers do: suggesting solutions to error messages”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, pp. 1019–1028. ISBN: 9781605589299. DOI: 10.1145/1753326.1753478. URL: https://dl.acm.org/doi/10.1145/1753326.1753478.

[26] Emma Harvey, Allison Koenecke, and Rene F. Kizilcec. “”Don’t Forget the Teachers”: Towards an Educator-Centered Understanding of Harms from Large Language Models in Education”. In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. CHI ’25. Association for Computing Machinery, 2025. ISBN: 9798400713941. DOI: 10.1145/3706598.3713210. URL: https://dl.acm.org/doi/10.1145/3706598.3713210.

[27] Anja Hawlitschek et al. “Drop-out in programming courses – prediction and prevention”. In: *Journal of Applied Research in Higher Education* ahead-of-print (July 2019). DOI: 10.1108/JARHE-02-2019-0035.

[28] Jia He et al. *Does Prompt Formatting Have Any Impact on LLM Performance?* 2024. arXiv: 2411.10541 [cs.CL]. URL: https://arxiv.org/abs/2411.10541.

[29] Xinyi Hou et al. “Large language models for software engineering: A systematic literature review”. In: *ACM Transactions on Software Engineering and Methodology* 33.8 (2024), pp. 1–79.

[30] Maria Hristova et al. “Identifying and correcting Java programming errors for introductory computer science students”. In: *SIGCSE Bull.* 35.1 (Jan. 2003), pp. 153–156. ISSN: 0097-8418. DOI: 10.1145/792548.611956. URL: https://dl.acm.org/doi/10.1145/792548.611956.

[31] Paul Hudak. “Conception, evolution, and application of functional programming languages”. In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 359–411. ISSN: 0360-0300. DOI: 10.1145/72551.72554. URL: https://dl.acm.org/doi/10.1145/72551.72554.

[32] John Hughes. “Why functional programming matters”. In: *The computer journal* 32.2 (1989), pp. 98–107.

[33] Binyuan Hui et al. *Qwen2.5-Coder Technical Report*. 2024. arXiv: 2409.12186 [cs.CL]. URL: https://arxiv.org/abs/2409.12186.

[34] Naman Jain et al. *LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code*. 2024. arXiv: 2403.07974 [cs.SE]. URL: https://arxiv.org/abs/2403.07974.

[35] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[36] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770 [cs.CL]. URL: https://arxiv.org/abs/2310.06770.

[37] Sungmin Kang et al. *Explainable Automated Debugging via Large Language Model-driven Scientific Debugging*. 2023. arXiv: 2304.02195 [cs.SE]. URL: https://arxiv.org/abs/2304.02195.

[38] Nam Ju Kim, Brian R Belland, and Andrew E Walker. “Effectiveness of computer-based scaffolding in the context of problem-based learning for STEM education: Bayesian meta-analysis”. In: *Educational Psychology Review* 30.2 (2018), pp. 397–429.

[39] David R Krathwohl. “A revision of Bloom’s taxonomy: An overview”. In: *Theory into practice* 41.4 (2002), pp. 212–218.

[40] John Lee, Fengkai Liu, and Tianyuan Cai. “Code Debugging with LLM-Generated Explanations of Programming Error Messages”. In: Nov. 2024, pp. 1–5. DOI: 10.1109/ICEED62316.2024.10923833.

[41] Juho Leinonen et al. “Using Large Language Models to Enhance Programming Error Messages”. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2023. Toronto ON, Canada: Association for Computing Machinery, 2023, pp. 563–569. ISBN: 9781450394314. DOI: 10.1145/3545945.3569770. URL: https://dl.acm.org/doi/10.1145/3545945.3569770.

[42] Jiawei Liu et al. *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. 2023. arXiv: 2305.01210 [cs.SE]. URL: https://arxiv.org/abs/2305.01210.

[43] Yilun Liu et al. “Interpretable Online Log Analysis Using Large Language Models with Prompt Strategies”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC ’24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 35–46. ISBN: 9798400705861. DOI: 10.1145/3643916.3644408. URL: https://dl.acm.org/doi/10.1145/3643916.3644408.

[44] Katherine L McNeill and Dean M Martin. “Claims, evidence, and reasoning”. In: *Science and Children* 48.8 (2011), p. 52.

[45] Alexandru-Radu Moraru. *Project Replication Packages*. https://github.com/alemoraru/exceed-project-overview. 2025.

[46] Ike Obi et al. “Identifying Factors Contributing to Bad Days for Software Developers: A Mixed Methods Study”. In: *arXiv preprint arXiv:2410.18379* (2024).

[47] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774.

[48] Michael Perscheid et al. “Studying the advancement in debugging practice of professional software developers”. In: *Software Quality Journal* 25.1 (Mar. 2017), pp. 83–110. ISSN: 0963-9314. DOI: 10.1007/s11219-015-9294-2. URL: https://dl.acm.org/doi/10.1007/s11219-015-9294-2.

[49] James Prather et al. “The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers”. In: *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*. ICER ’24. Melbourne, VIC, Australia: Association for Computing Machinery, 2024, pp. 469–486. ISBN: 9798400704758. DOI: 10.1145/3632620.3671116. URL: https://dl.acm.org/doi/10.1145/3632620.3671116.

[50] Christian Prause, Ralf Gerlich, and Rainer Gerlich. “Fatal Software Failures in Spaceflight”. In: *Encyclopedia* 4 (June 2024), pp. 936–965. DOI: 10.3390/encyclopedia4020061.

[51] David Pritchard. “Frequency distribution of error messages”. In: *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 1–8. ISBN: 9781450339070. DOI: 10.1145/2846680.2846681. URL: https://dl.acm.org/doi/10.1145/2846680.2846681.

[52] Meghana Rajeev et al. *Cats Confuse Reasoning LLM: Query Agnostic Adversarial Triggers for Reasoning Models*. 2025. arXiv: 2503.01781 [cs.CL]. URL: https://arxiv.org/abs/2503.01781.

[53] Audrey Salmon et al. *Debugging Without Error Messages: How LLM Prompting Strategy Affects Programming Error Explanation Effectiveness*. 2025. arXiv: 2501.05706 [cs.SE]. URL: https://arxiv.org/abs/2501.05706.

[54] Eddie Antonio Santos and Brett A. Becker. “Not the Silver Bullet: LLM-enhanced Programming Error Messages are Ineffective in Practice”. In: *Proceedings of the 2024 Conference on United Kingdom & Ireland Computing Education Research*. UKICER ’24. Manchester, United Kingdom: Association for Computing Machinery, 2024. ISBN: 9798400711770. DOI: 10.1145/3689535.3689554. URL: https://dl.acm.org/doi/10.1145/3689535.3689554.

[55] Eddie Antonio Santos, Prajish Prasad, and Brett A. Becker. “Always Provide Context: The Effects of Code Context on Programming Error Message Enhancement”. In: *Proceedings of the ACM Conference on Global Computing Education Vol 1*. CompEd 2023. Hyderabad, India: Association for Computing Machinery, 2023, pp. 147–153. ISBN: 9798400700484. DOI: 10.1145/3576882.3617909. URL: https://dl.acm.org/doi/10.1145/3576882.3617909.

[56] Anton Semenkin and Michelle Frost. *Mellum Goes Open Source: A Purpose-Built LLM for Developers, Now on Hugging Face*. JetBrains AI Blog. https://blog.jetbrains.com/ai/2025/04/mellum-goes-open-source-a-purpose-built-llm-for-developers-now-on-hugging-face/. Apr. 2025.

[57] Atsushi Shirafuji et al. “Rule-Based Error Classification for Analyzing Differences in Frequent Errors”. In: Nov. 2023, pp. 1–7. DOI: 10.1109/TALE56641.2023.10398341.

[58] John Sweller. “Cognitive load theory”. In: *Psychology of learning and motivation*. Vol. 55. Elsevier, 2011, pp. 37–76.

[59] Alex Tamkin et al. *Understanding the Capabilities, Limitations, and Societal Impact of Large Language Models*. 2021. arXiv: 2102.02503 [cs.CL]. URL: https://arxiv.org/abs/2102.02503.

[60] Mohsen Tavakol and Reg Dennick. “Making sense of Cronbach’s alpha”. In: *International journal of medical education* 2 (2011), p. 53.

[61] Andrew Taylor et al. “dcc –help: Transforming the Role of the Compiler by Generating Context-Aware Error Explanations with Large Language Models”. In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 1314–1320. ISBN: 9798400704239. DOI: 10.1145/3626252.3630822. URL: https://dl.acm.org/doi/10.1145/3626252.3630822.

[62] CodeGemma Team et al. *CodeGemma: Open Code Models Based on Gemma*. 2024. arXiv: 2406.11409 [cs.CL]. URL: https://arxiv.org/abs/2406.11409.

[63] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2025. arXiv: 2312.11805 [cs.CL]. URL: https://arxiv.org/abs/2312.11805.

[64] Emillie Thiselton and Christoph Treude. *Enhancing Python Compiler Error Messages via Stack Overflow*. 2019. arXiv: 1906.11456 [cs.SE]. URL: https://arxiv.org/abs/1906.11456.

[65] Pablo Valenzuela-Toledo et al. *Explaining GitHub Actions Failures with Large Language Models: Challenges, Insights, and Limitations*. 2025. arXiv: 2501.16495 [cs.SE]. URL: https://arxiv.org/abs/2501.16495.

[66] Sierra Wang, John Mitchell, and Chris Piech. "A Large Scale RCT on Effective Error Messages in CS1". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 1395–1401. ISBN: 9798400704239. DOI: 10.1145/3626252.3630764. URL: https://dl.acm.org/doi/10.1145/3626252.3630764.

[67] J. Christopher Westland. "The cost of errors in software development: evidence from industry". In: *J. Syst. Softw.* 62.1 (May 2002), pp. 1–9. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(01)00130-3. URL: https://dl.acm.org/doi/10.1016/S0164-1212(01)00130-3.

[68] Patricia Widjojo and Christoph Treude. *Addressing Compiler Errors: Stack Overflow or Large Language Models?* 2023. arXiv: 2307.10793 [cs.SE]. URL: https://arxiv.org/abs/2307.10793.

[69] John Yang et al. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. 2024. arXiv: 2405.15793 [cs.SE]. URL: https://arxiv.org/abs/2405.15793.

[70] Stephanie Yang et al. "Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions". In: *ACM Trans. Comput. Educ.* 24.4 (Nov. 2024). DOI: 10.1145/3690652. URL: https://dl.acm.org/doi/10.1145/3690652.

[71] Yang Zhang et al. "How do Developers Talk about GitHub Actions? Evidence from Online Software Development Community". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3623327. URL: https://dl.acm.org/doi/10.1145/3597503.3623327.

[72] Zhehao Zhang et al. *Personalization of Large Language Models: A Survey*. 2025. arXiv: 2411.00027 [cs.CL]. URL: https://arxiv.org/abs/2411.00027.

[73] Michael Zhivich and Robert K. Cunningham. "The Real Cost of Software Errors". In: *IEEE Security Privacy* 7.2 (2009), pp. 87–90. DOI: 10.1109/MSP.2009.56.

[74] Jeffrey Zhou et al. *Instruction-Following Evaluation for Large Language Models*. 2023. arXiv: 2311.07911 [cs.CL]. URL: https://arxiv.org/abs/2311.07911.

[75] Jingming Zhuo et al. *ProSA: Assessing and Understanding the Prompt Sensitivity of LLMs*. 2024. arXiv: 2410.12405 [cs.CL]. URL: https://arxiv.org/abs/2410.12405.

A

Python Skill Level Assessment Qualtrics Questionnaire and Details

This appendix collects material that was used for, or is directly relevant to, the Python Skill Level Assessment that informed our pilot study, which we launched on Qualtrics[1]. It brings together the elements that shaped the assessment and the specific items that we later relied on in the main study[2].

Table A.1 presents the self rating question for Python experience that we used in the pilot study. The question uses six levels that are informed by the Dreyfus model of skill acquisition. The table shows the exact wording of the question and the response options so that the reader can see how participants reported their level of experience.

Figure A.1 shows the eight multiple choice questions that we selected from an initial pool of 56 items[3]. These eight items were identified as the most capable of distinguishing novices from experts in the context of debugging and understanding error messages. As explained in Section 3.1, this set of eight items was then carried forward and used in the main Prolific study.

Numerical Level	Experience level statement
1	I have no previous experience in Python.
2	Focusing on syntax & basics, relying on tutorials, lacking real-world project experience.
3	Practical experience, small projects or assignments. Grasping basic concepts, and troubleshooting independently.
4	Experience with projects. Able to independently plan and execute tasks, proficient with Python's libraries.
5	Experience in complex projects. Deep understanding, ability to consciously resolve difficult problems
6	Significant experience in larger complex programs. Solves complex problems effortlessly and subconsciously

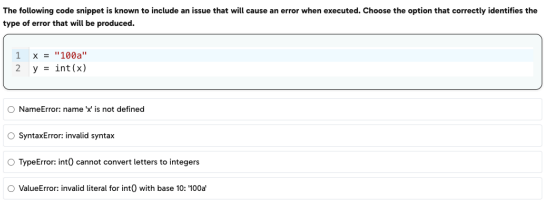
Table A.1: Six-level self-rating of Python experience used in the study, informed by the Dreyfus skill-acquisition model.

[1] https://www.qualtrics.com/

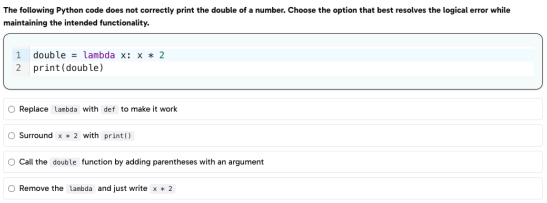
[2] Readers ought to remember that within the main contents of this paper, Section 3.1 describes the motivation for creating the assessment and the approach we followed during the pilot phase. It explains why a dedicated instrument was needed and how we constructed and refined it.

[3] For brevity, we do not include all 56 pilot questions here. The full set, along with additional plots, and the analysis scripts and notebooks used to select the final eight, is available at https://github.com/alemoraru/exceed-python-skill-assessment.

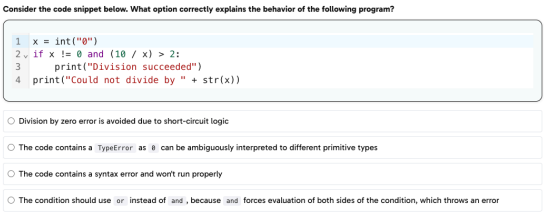
(a) Question ID Q4.6 of the Python Qualtrics Skill-Assessment Survey.



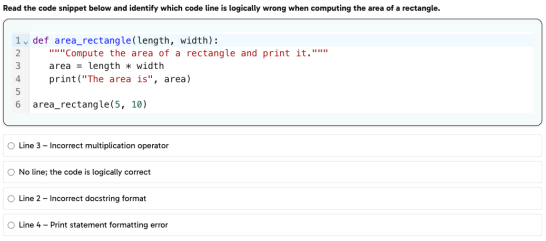
(b) Question ID Q5.2 of the Python Qualtrics Skill-Assessment Survey.



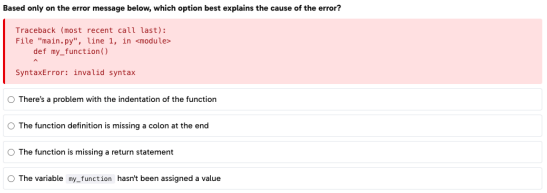
(c) Question ID Q6.5 of the Python Qualtrics Skill-Assessment Survey.



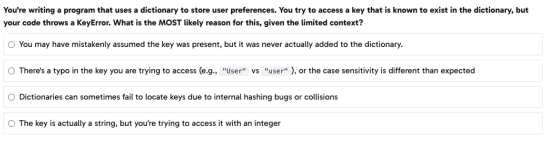
(d) Question ID Q7.1 of the Python Qualtrics Skill-Assessment Survey.



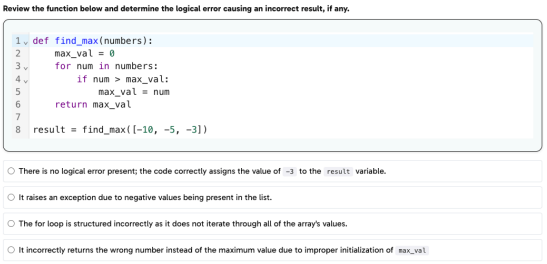
(e) Question ID Q7.2 of the Python Qualtrics Skill-Assessment Survey.



(f) Question ID Q8.2 of the Python Qualtrics Skill-Assessment Survey.



(g) Question ID Q9.5 of the Python Qualtrics Skill-Assessment Survey.



(h) Question ID Q10.1 of the Python Qualtrics Skill-Assessment Survey.

Figure A.1: Final set of survey items that most effectively discriminated between novices and experts in the Python Skill-Assessment Survey. The images reproduce the question options and interface, as presented in the Prolific web application during the main study.

B

Fantastic Four Code Snippets and Error Messages

This appendix presents the four buggy Python snippets used in the Prolific study, with their standard interpreter errors. Sections B.1–B.4 show each code snippet followed by its error message.

B.1. Snippet A: `SyntaxError: unterminated string literal`

```
1  class BookShelf:
2      def __init__(self, log):
3          self.log = log
4
5      def preview(self):
6          """
7          Shows first two book titles, if any.
8          """
9          if not os.path.exists(self.log):
10             return "No␣log␣found."
11         with open(self.log) as f:
12             lines = f.readlines()
13         preview = "".join(lines[:2])
14         return f"""Preview:\n{preview}""
15
16     def summary(self):
17         """
18         Gives a summary of the log.
19         """
20         total = count_books(self.log)
21         return f"Books logged: {total}"
22
23 def add_book(log, title):
24     """
25     Adds a book entry with timestamp.
26     """
27     with open(log, 'a') as f:
28         f.write(f"{datetime.now().isoformat()} - {title}\n")
29
30 def count_books(log):
31     """
32     Counts books in the log.
33     """
34     if not os.path.exists(log):
35         return 0
36     with open(log) as f:
37         return sum(1 for _ in f)
```

Listing B.1: One of the chosen four buggy Python code snippet, which when executed as a standalone file, will trigger a `SyntaxError`, due to the unclosed triple-quoted string literal on line 17.

```
File "main.py", line 36
  """
  ^
SyntaxError: unterminated triple-quoted string literal
(detected at line 40)
```

Figure B.1: Standard Python interpreter message displayed when running the code in Listing B.1.

B.2. Snippet B: `NameError: name not defined`

```python
import random


class UserData:
    """Represents user data with a name and a list of scores."""

    def __init__(self, name, scores):
        self.name = name
        self.scores = scores

    def top_score(self):
        """Returns the highest score."""
        return maximum(self.scores) if self.scores else 0

    def add_score(self, score):
        self.scores.append(score)


def summarize_scores(users):
    return {u.name: u.top_score() for u in users}


if __name__ == '__main__':
    """Main routine to generate user data, summarize scores, and print the results."""
    users = [UserData(f"user_{i␣+␣1}", [random.randint(0, 100) for _ in range(random.randint
        (2, 5))]) for i in range(4)]
    summary = summarize_scores(users)
    for name, score in summary.items():
        print(f"{name}:␣{score:.2f}")
```

Listing B.2: One of the chosen four buggy Python code snippet, which when executed as a standalone file, will trigger a `NameError` on line 13 of the code listing, due to the undefined variable `maximum`.

```
Traceback (most recent call last):
  File "main.py", line 26, in <module>
    summary = summarize_scores(users)
              ^^^^^^^^^^^^^^^^^^^^^^^
  File "main.py", line 20, in summarize_scores
    return {u.name: u.top_score() for u in users}
                    ^^^^^^^^^^^^
  File "main.py", line 13, in top_score
    return maximum(self.scores) if self.scores else 0
           ^^^^^^^
NameError: name 'maximum' is not defined
```

Figure B.2: Standard Python interpreter message displayed when running the code in Listing B.2.

B.3. Snippet C: TypeError: unsupported operand types

```python
import random

def generate_scores(n):
    """Generate n random test scores."""
    return [random.randint(0, 100) for _ in range(n)]

def average(scores):
    """Compute the average score."""
    if not scores:
        return 0
    return sum(scores) / len(scores)

def filter_passing(scores, threshold=60):
    """Return scores that are passing."""
    return [s for s in scores if s >= threshold]

class ScoreReport:
    def __init__(self, scores):
        self.scores = scores

    @classmethod @staticmethod
    def describe():
        """Describe the scoring system."""
        return "Scores range from 0 to 100."

    def passing_percentage(self):
        """Return the percentage of passing scores."""
        passing = filter_passing(self.scores)
        return 100 * len(passing) / len(self.scores) if self.scores else 0

    def report(self):
        """Return a formatted report."""
        avg = average(self.scores)
        pct = self.passing_percentage()
        desc = self.describe()
        return f"{desc}\nAverage: {avg:.1f}\nPassing: {pct:.1f}%"

def main():
    scores = generate_scores(12)
    report = ScoreReport(scores)
    print(report.report())

if __name__ == "__main__":
    main()
```

Listing B.3: One of the chosen four buggy Python code snippet, which when executed as a standalone file, will trigger a `TypeError` on line 21 of the code listing, due to adding both `@classmethod` and `@staticmethod` on the same line.

```
Traceback (most recent call last):
  File "main.py", line 17, in <module>
    class ScoreReport:
  File "main.py", line 21, in ScoreReport
    @classmethod @staticmethod
     ~~~~~~~~~~~~^~~~~~~~~~~~~
TypeError: unsupported operand type(s) for @: 'type' and 'type'
```

Figure B.3: Standard Python interpreter message displayed when running the code in Listing B.3.

B.4. Snippet D: TypeError: object is not subscriptable

```python
import math

def normalize(vec):
    norm = math.sqrt(sum(x ** 2 for x in vec))
    return [x / norm for x in vec] if norm else vec

def dot(a, b):
    return sum(x * y for x, y in zip(a, b))

def cosine(a, b):
    if len(a) != len(b): raise ValueError("Vectors must be of the same length")
    return dot(normalize(a), normalize(b))

def fixed_vectors():
    """Returns a fixed set of vectors for testing purposes."""
    return [
        [1.0, 2.0, 3.0],
        [2.0, 0.0, 1.0],
        [-1.0, 1.0, 0.0],
        [0.5, -2.0, 2.0]
    ]

def most_similar_pair(vectors):
    """Finds the most similar pair of vectors based on cosine similarity."""
    max_sim = -2
    pair = (0, 1)
    for i in range(len(vectors)):
        for j in range(i + 1, len(vectors)):
            sim = cosine(vectors[i], vectors[j])
            if sim > max_sim:
                max_sim = sim
                pair = (i, j)
    return pair

def main():
    vs = fixed_vectors()
    print("Most similar pair:", most_similar_pair(vs))
    for i in range(len(vs)):
        print("Vector", i, ":", vs.__getitem__[i])

if __name__ == "__main__":
    main()
```

Listing B.4: One of the chosen four buggy Python code snippet, which when executed as a standalone file, will trigger a `TypeError` on line 39 of the code listing, due to using the wrong syntax for accessing a list's elements.

```
Traceback (most recent call last):
  File "main.py", line 42, in <module>
    main()
  File "main.py", line 39, in main
    print("Vector", i, ":", vs.__getitem__[i])
                            ~~~~~~~~~~~~~~^^^
TypeError: 'builtin_function_or_method' object is not subscriptable
```

Figure B.4: Standard Python interpreter message displayed when running the code in Listing B.4.

C

Prolific Full-Stack Web Application Design Details

This appendix describes the full-stack platform we deployed on Prolific for the main study and explains the key design decisions behind it. We coordinated services with Docker Compose[1] to keep components modular and easy to start, stop, and update. Containerization also allowed us to deploy everything on a Linux virtual machine with only the `docker` binary installed. This reduced configuration drift over time as we developed the application, and improved reproducibility, which was an important requirement for our study.

C.1. System Architecture

The platform consists of a web frontend, a Python backend, a relational database, a local LLM inference service, and a reverse proxy. Figure C.1 shows the high-level architecture and how these components interact within a containerized environment.

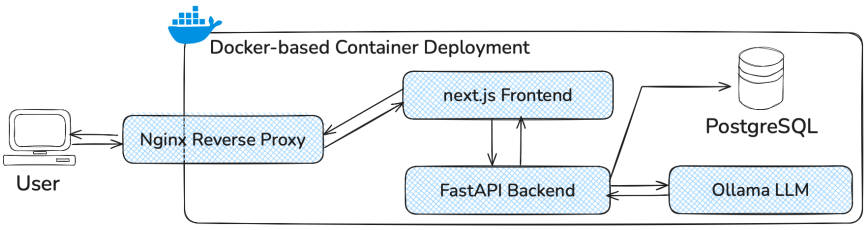


Figure C.1: System architecture of the Prolific study full-stack application: containerized frontend, backend, database, local LLM inference, and reverse proxy orchestrated with Docker Compose. We coordinated everything using a single GitHub repository, https://github.com/alemoraru/exceed-prolific-orchestrator.

The frontend[2] was implemented with Next.js, React, Typescript, and TailwindCSS. We chose this stack mostly due to familiarity and because it was straightforward to build reusable components, such as forms, multi-step submission flows, and interactive elements for task execution and feedback collection.

The backend[3] was built with FastAPI[4] in Python. FastAPI offers strong type support, automatic API documentation, and efficient asynchronous I/O. These features suited our needs for handling user-provided code execution and returning LLM-rephrased error messages. Keeping the backend in Python also simplified integration with our evaluation logic and the LLM client used for inference.

[1] https://docs.docker.com/compose/
[2] https://github.com/alemoraru/exceed-prolific-frontend
[3] https://github.com/alemoraru/exceed-prolific-backend
[4] https://fastapi.tiangolo.com/

We stored participant data, submissions, feedback, and event logs in PostgreSQL. PostgreSQL provides mature tooling, transactional guarantees, and expressive queries. A relational schema with ACID properties helped maintain integrity across multi-step tasks, for example when linking code submissions to feedback and timing events. Given our relatively modest scale, any other relational or non-relational database would have been equally suitable for our requirements, yet PostgreSQL aligned best with our planned analysis workflow.

Local LLM inference was provided by Ollama. Running inference on a self-hosted service gave us control over data handling and latency while not incurring additional costs by using an external API[5], such as OpenAI's API. The service loaded the model used for error-message rephrasing. Our application can technically support any other open-source models available on Ollama within the available memory and latency budget. However, as described in Section 3.3, we selected `llama-3.1-8B-Instruct`.

Finally, Nginx[6] served as the reverse proxy. It routed requests to the frontend and backend and cached static assets to improve responsiveness during data collection. It also restricted public access to a single entry point, which simplified networking, while reducing exposure and risks for seeing malicious activity.

C.2. Deployment and Networking

All services ran as containers managed by Docker Compose. Only Nginx was exposed to the outside network on port 80. The backend, database, and LLM service communicated over an internal Docker network, which reduced the attack surface and allowed services to address each other by name. The frontend interacted with the backend through proxied API routes, and static assets benefited from Nginx caching. This layout kept the public interface relatively secure while allowing internal services to restart or scale independently. In practice, load was very low and no scaling actions were even required.

Runtime configuration was provided through environment variables. Some of the settings included database credentials and the database URL for the backend, the model identifier for the LLM service, and the frontend host URL used for API requests. For our research deployment, however, we used a set of conventional defaults, as we did not need to deploy on multiple environments with different settings.

C.3. Monitoring and Inspection

During development and data collection we used a small Python utility, `postgres/db_inspect.py`, to query selected tables and verify that submissions, feedback, and events were recorded as expected. The script also supported troubleshooting ingestion issues and exporting snapshots of collected data. These checkpoints helped us monitor progress, assess data quality, and decide whether to recruit additional participants when attrition or exclusion, due to malicious behavior, affected our stratified sampling procedure.

[5] https://openai.com/api/
[6] https://nginx.org/