Comparing the hint quality of a Small Language Model and a Large Language Model in automatic hint generation

Replacing the LLM inside the JetBrains Academy AI hint generation system with a RAG-augmented SLM

Casper Dekeling[1,3]
Supervisor(s): Gosia Migut[1]**, Anastasiia Birillo**[2]
[1]EEMCS, Delft University of Technology, The Netherlands
[2]JetBrains, Belgrade, Serbia
[3]JetBrains, Amsterdam, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Master of Computer Science
June 10, 2025

Name of the student: Casper Dekeling
Final project course: IN5000
Thesis committee: Marcus Specht[1], Arie van Deursen[1], Gosia Migut[1]

Contents

I. Introduction

Generative Artificial Intelligence (AI) and Natural Language Processing (NLP) have been ever-growing topics, especially since the success of ChatGPT in 2022. Now, ChatGPT and other generative AI chatbots, like Gemini, Claude and DeepSeek are being used by many people in their daily lives, from answering general questions to writing code.

While the use of AI in education is still a hotly debated topic, there are areas where students can benefit from the use of AI. One of these situations is automatic hint generation. If a student is stuck on an assignment or a piece of theory, a student might not be able to get the help they need due to challenges such as a growing student-to-teacher ratio (10), the need to ask questions outside of regular working hours, or not feeling comfortable seeking help from the staff (11). However, if we reinforce the staff with an AI tutor that can help students 24/7, by giving them hints to guide them, this problem could be alleviated (7; 8; 9; 12). Therefore, numerous such systems have been developed that make use of LLMs like ChatGPT to generate hints and guide students when they need help.

When using AI for anything, there are always concerns that must be taken into account. Because of the probabilistic nature of AI, there will always be a lack of control over the output (1), which is magnified if the AI used is a top-of-the-line model like ChatGPT, which is controlled and hosted by a commercial company, and not by the organisation using it. It also creates privacy concerns, as the questions, prompts and data you send to the AI can be used for other purposes by the company that hosts the AI. Lastly, computationally intensive AI models cause environmental and financial concerns, as they are usually run on powerful, energy-draining hardware and cost money to use (4; 5). These problems can be mitigated if we swap the Large Language Model (LLM) for a Small Language Model (SLM) instead: a scaled-down model, which massively reduces its size, while keeping the output quality as high as possible. This allows the smaller model to be run on commercially available hardware, and even provides the possibility to fine-tune and tweak the model to fit the purpose for which it will be used.

However, smaller models are not without drawbacks, as the output quality does drop when the model's size is reduced (6). There are, however, ways to minimise the drop in output

quality, such as giving the SLM access to an external knowledgebase by using Retrieval Augmented Generation (RAG). The question then becomes whether SLMs are *good enough* to function in an AI hint-generation system, which is what this research focuses on. Our research questions are:

- How does replacing an LLM with an SLM as part of an AI hint-generation system affect the hint quality, as determined by experts?
- How does replacing an LLM with an SLM as part of an AI hint-generation system affect student satisfaction and trust in the system?

The novelty of this research lies firstly in the niche domain in which the SLM-system will be tested: the Kotlin programming language. Kotlin is a much less prominent programming language than something like Python, and would thus be less prevalent in the SLM's training data. We also combine an expert experiment with a student experiment to have a broader overview of how certain criteria for hint quality relate to student satisfaction and trust in the system. Lastly, in the use case for this research, the SLM is part of a larger system that includes static analysis and static processing of the generated hints, possibly further reducing drawbacks that might be caused by replacing an LLM with an SLM.

This thesis includes preliminary materials, presented in section II, which explain the concepts important to this research, such as LLMs, SLMs and RAG and the theory behind them. Following this is section III, which presents the scientific article, containing only the core of the research. The material in section II thus aims to give the reader all the necessary knowledge to fully understand the article in section III.

II. Preliminary materials

II.1. Natural Language Processing

Natural Language Processing (NLP) is a subfield of Artificial Intelligence that focuses on allowing machines to understand, interpret, and generate natural language. This includes a wide range of tasks, such as text classification or sentiment analysis, machine translation, question answering, summarisation, and even generating more complex outputs like code or mathematical proofs.

A fundamental aspect of NLP is how words are understood internally by the machine. First, text is split into **tokens**, which can be words, subwords or characters. For example, "unbelievable" might be split up into "un", "believe" and "able", each with a different contribution to the definition of the entire word. The model will then try to understand each token individually in such a way that it will also understand that the "un" in "unforgettable" has the same meaning as in "unbelievable". It does this by creating **embeddings** for each token, which are vectors of floating-point numbers, representing the meaning of a (sub-)word. The vectors are learned during training and capture both syntactic information (verb/noun) and semantic meaning (e.g. "apple" is close to "pear"). Each element of the vector represents a characteristic of the word. If we take the domain of animals as an example, an embedding vector could correspond to a list of characteristics such as *size, hairiness, speed, strength,* and *danger*. The model is then trained to learn the characteristics of all animals and can make inferences based on them. For example, if it is given the word "dog", and tasked with finding a more dangerous version of a dog, it would simply find the "dog" vector in the vector space, and move in the direction of increasing danger, where it would find words like "wolf". Thus, it can generate the word "wolf" based on the two pieces of information that it is looking for a dog, but more dangerous. See figure I for a visualisation of such embeddings.

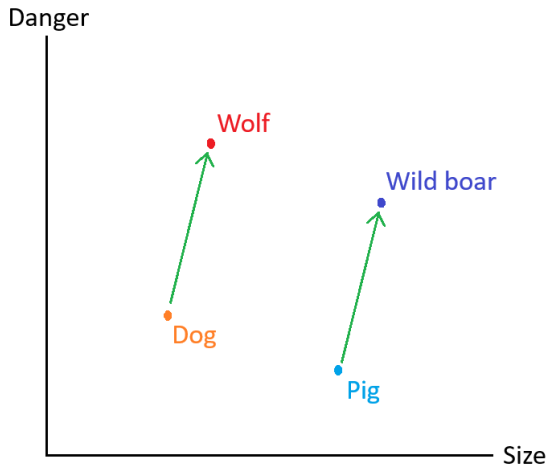

Figure I: A 2D example of animal examples. While a pig is completely different from a dog, and a wild boar is completely different from a wolf, the *difference* between a wild boar and a pig, is the same as the *difference* between a wolf and a dog.

Once a model understands the words in its vocabulary, generating text is simply an estimation of which token is likely to be next, based on the prompt it received and the tokens it has already generated. The next token is then sampled from a probability distribution, in one of many different ways, such as greedy (always pick the most likely token), top-k (sample

from the k most likely tokens) or using a threshold (sample from all words with a probability more than p). This process then repeats for the next token. Thus, text generation will always be probabilistic, and asking the model the exact same thing multiple times might result in completely different responses.

As an example for prompt completion, take the prompt "I am hungry so I". The most likely tokens to appear next might be "ate" or "ordered". Say it samples the word "ate", the model then continues by completing the prompt "I am hungry so I ate", where the most likely next tokens might be "spaghetti", "pancakes", or the token that signifies the end of a sentence or prompt. However, if it sampled "ordered", the most likely next token might be "Domino's" or "McDonalds", already resulting in a completely different probability distribution for the next token.

II.2. Transformers

What makes the complex calculations in NLP possible is the transformer architecture. The key principle in the transformer architecture is **attention**. Every time the transformer looks at a certain token, it also looks at the entire text around this token and extracts the other relevant parts from that text. For example, in the sentence "The chicken crossed the road, because it wanted to get to the other side.", when the transformer is looking at the token "it", it will consider the "chicken" token relevant, but "road" and "side" less relevant. After the *attention layer*, the result for each token is passed through a neural network, transforming it into another intermediate result.

The attention calculation, followed by a small neural network can be seen as one **layer** of the transformer. The transformer consists of many layers, where each layer uses the output of the previous layer as its input. Thus, the transformer can look at an entire piece of text, instead of just small parts of it. Lower layers might detect basic grammar, but higher layers will detect topics, tone or structure in the entire text.

In essence, a layer is just a sequence of complex matrix multiplications, which means the matrices that are multiplied with the input to generate the output are the *parameters* of the model.

II.2.1. Large Language Models

At a high level, Large Language Models (LLMs) like ChatGPT are deep neural networks based on the transformer architecture. They consist of an embedding layer to map tokens to embeddings, transformer layers to model dependencies within the text, and an output layer that predicts the next token. The models are trained on massive datasets, and can be made up of hundreds of billions, even trillions of parameters. To simply store an LLM like ChatGPT you would need hundreds of gigabytes, possibly terabytes of storage, let alone the computational power required to run the model.

II.2.2. Small Language Models

To make it feasible to run a model on commercially available hardware, LLMs can be shrunk down to Small Language Models (SLMs). There are several techniques to accomplish this, such as training a "student" SLM model to mimic the outputs of a larger "teacher" LLM model. Alternatively, an LLM model can be pruned, removing redundant or less important weights/parameters from the network. It can even be as simple as reducing the numerical precision of parameters, changing them from 32-bit floats to 8-bit floats. A smart combination of techniques, followed by fine-tuning the model, can bring the model size down to a range between a billion to a hundred billion parameters. The smaller SLMs can even be run on somewhat powerful consumer laptops. However, the loss of parameters and precision also results in a decrease in the output quality. The model has less internal knowledge and reasoning capabilities, thus resulting in knowledge gaps or hallucinations. This is a trade-off that must be considered, though it can be diminished by enhancing the SLM with techniques such as Retrieval Augmented Generation.

II.3. Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a technique to augment a model with an external knowledgebase, to cover gaps in the model's internal knowledge. It can be divided into two parts: the knowledgebase, and the retriever.

The knowledgebase can be any collection of documents, from code documentation to med-

ical files. To make the knowledgebase usable for the retriever, it must first be vectorised. This means we will divide the documents into chunks and convert each chunk into embeddings. The vectors are then stored in a special vector storage database, designed to be efficient in storing and retrieving vectors. The vectorised database will be where the retriever can search through all documents.

Once a request to the model is made, the retriever will find documents from the knowledgebase that might be relevant for the request, before it is sent to the model. It does so by first converting the request into embeddings. Now, it can compare the embeddings obtained from the request with the embeddings in the vector storage. It will find all documents that have content similar to the request, by using a similarity metric such as cosine distance. Since we are working with embeddings, it goes beyond simply searching all documents for words that are also used in the request, and instead finds all documents where any information is similar in meaning to information from the request. The retriever will generate a score for every document, and then returns a set of documents based on a method like *top-k* (return the k highest-scoring documents), or a *threshold* (return all documents with a score higher than the threshold). The retrieved documents will then be combined with the original request, which is then passed to the model. Thus, until that point, the model itself has done nothing with the request.

RAG is especially beneficial in smaller models, where the model's smaller size means it can memorise less information, contrary to massive LLMs, which can easily store a lot of information in its internal (parametric) memory. RAG can thus be used to reduce the knowledge gap between SLMs and LLMs. Other benefits of RAG are that you can completely design the knowledgebase to match the purpose of the model, giving the model access to the right information, even if the domain is niche. Additionally, the knowledgebase can easily be replaced and updated, in contrast to a model's parametric memory. Thus, augmenting your model with RAG makes it more flexible and future-proof.

II.4. Prompt engineering

The bridge between the user and the AI, is the prompt the user sends to the AI. Therefore, the prompt must be as clear as possible and communicate clearly what the user expects from

the model's response. Designing the optimal prompt for a user's purpose is called prompt engineering. This section explains common prompt engineering techniques.

II.4.1. Few-shot learning

For simple tasks, asking the AI a simple question will be enough to get a satisfactory response. However, some tasks might have more specific requirements for the response, such as what is included in the response or how the response is structured. In this case, you can teach the AI to respond in this way within your prompt by applying few-shot learning. Few-shot learning entails extending your prompt with examples of how you want the AI to respond. For example, when asking for a hint on a programming exercise, you could include other exercises, and examples of how you would want the AI to respond in those contexts. Thus, the model has a clear template, which allows for more personalised responses.

II.4.2. Chain-of-thought

Another technique to improve the quality of the AI's output is to force it to reason about its answer step by step. By asking the AI to provide step-by-step reasoning for its response, it is forced to support every step in its response with logical reasoning, thus reducing the chance of hallucinations or misinformation. Chain-of-thought prompting is especially useful for more complex or multi-step problems.

An alternative way of approaching chain-of-thought reasoning would be to split up the prompt into multiple prompts. For example, when stuck on a coding task, before providing the code to the AI, you could first prompt it to generate a list of subgoals for the assignment, giving a step-by-step overview of how to solve the assignment. Then, you could provide your code, together with the generated list of subgoals and ask which subgoals have been completed and what needs to be done to complete the next subgoal on the list. By making the model use its own subgoals, you are forcing it to reason methodically about the problem.

II.4.3. Clear and concise prompts

Another important factor in creating an effective prompt is not to include redundant information. Doing so could cause the AI to focus on the wrong thing, and for smaller models,

especially, could prevent the model from seeing dependencies in the text, because it has a smaller *context window*, meaning it can only see dependencies that are closer together in the prompt. Prompt clarity and conciceness is even more important when using a system that has been enhanced with RAG, as the RAG retriever will use the information from the prompt to find relevant documents. Thus, if there is irrelevant data in the prompt, it will retrieve irrelevant documents, which might make the AI focus on the wrong thing in its response.

Lastly, the structure of the prompt also plays an important role. As mentioned above, all models have a certain *context window*, and while it might be large in big LLMs, it can be a problem for smaller models if related information is separated and spread out far over the prompt. Thus, dependent information should be kept close together.

II.5. Our use case: JetBrains Academy extension

JetBrains has developed their own AI hint-generation system, integrated into their JetBrains Acadamy plugin (JBA) (2). The system allows students to click a 'Get Hint' button whenever they check their solutions and have failing tests. The system then uses the context inside the IDE (Integrated Development Environment, for example, IntelliJ) to ask the AI model for a next-step hint. The hint will be one single step towards a solution, guiding the student without giving the solution straightaway. The student is first presented with the hint in text form, describing what change needs to be made. If the text hint is not enough, the student can view the hint in their code, being presented with a code suggestion that the student can choose to accept or implement themselves.

Under the hood, the system is a lot more complicated than simply prompting an AI model. The system collects an extensive context from the IDE, including the student's progress and assignment information. The system has undergone extensive prompt engineering testing to create the most effective prompt. Lastly, the system applies static analysis to the generated hint to further improve its quality and have more control over the hint that is finally shown to the student. Currently, the model used is an LLM, more specifically ChatGPT-4o.

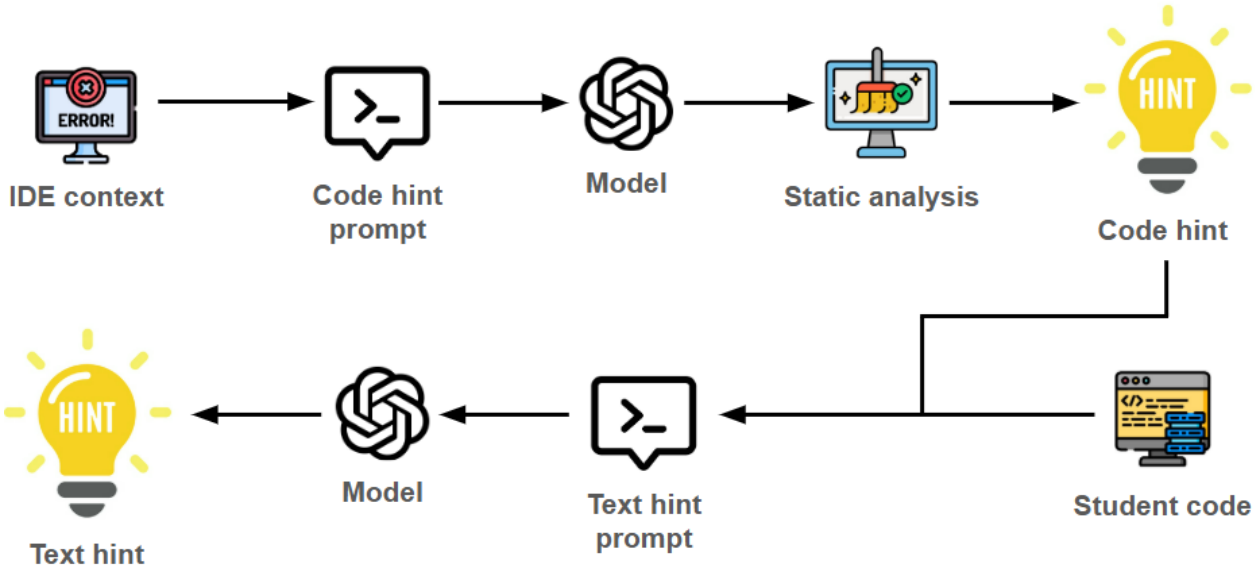

Figure II: Overview of the system flow for generating a code and text hint through the JetBrains AI hint generation system.

II.5.1. Flow of the system

When students follow a course through JBA, they can check their solution to a task through tests designed by the course author. If one or more tests fail, the student will receive feedback on the test that fails, and with the hint-generation system, will also get the option to press the 'Get Hint' button.

Once the 'Get Hint' button is pressed, the IDE will gather information from the student's solution and the assignment, format it as a prompt, and send the prompt to the AI model, which asks for a code hint to implement the next step towards a solution. Once the system receives the code hint, it applies static analysis to improve the quality of the hint.

With the final version of the code hint, the system sends a new prompt to the AI model, with the student's current code and the improved code from the code hint, and asks the model to describe the implemented changes textually. When the model responds with the text hint, the system first shows the text hint to students, only showing the code hint when the text hint is not enough for the student. See figure II for a visual overview of how the system generates both hints. Once it has generated both hints, it shows the text hint to the student and gives them the option to view the code hint. See figures A1 and A2 in appendix B for an in-IDE view of the system.

II.5.2. IDE Context

When the student asks for a hint, the system's first step is to gather an extensive context from the IDE. It gathers the **student's code changes** which they have made in the task to the assignment code, the **signatures of already implemented functions** in their code that may have been implemented in previous tasks, and lastly, it gathers their **test failure message(s)**. It also collects information from the assignment: the **task description** and the used **programming language**, **static hints** created by the course author, a list of **theory topics** relevant to the task, like "loops", **signatures of helper functions** that are used in the author's solution and the **String literals** used in the author's solution. The author's solution is not provided in its entirety to the model, to avoid bias towards one specific approach.

II.5.3. Static analysis

When the model generates the code hint, the system performs static analysis on the code hint to improve the quality and control what is shown to the student. The static analysis consists of multiple parts. First, irrelevant changes to the code are removed, such as when the model tries to improve the code quality of a function implemented in a previous task. This is achieved by comparing the student's solution to the author's solution and determining what functions should be changed or added, and which functions should not be.

Secondly, if a hint is generated for a function that is "short" in the author's solution, the author's complete implementation is used instead of the generated code hint. "Short" is defined as having at most 3 lines of function body, which is already considered an appropriate size for a next-step hint, and thus does not require any changes.

A crucial aspect of the hint quality is the code quality itself. To ensure proper code quality, the system uses the same code quality checks already part of JetBrains' IDEs to inspect and optimise the code generated by the model, thus fixing common mistakes and style issues.

Lastly, the biggest part of the static analysis is controlling the hint size. The model does not always understand that we are looking for a hint that only implements one step towards

the solution, and will sometimes generate code hints that implement large chunks of code at once. The code hint should correspond to one logical action, like creating a function without a body, or using a for loop, without a condition or body yet. To mitigate this, three heuristics are applied to the generated hint.

Firstly, if the first difference in the code is an addition to the code, only the first statement is retained. For example, if the change is creating a new function, including its body, only creating the function is kept, and the body is replaced with `TODO("Not yet implemented")`. If the change is a while-loop with a body, only the while loop is retained, with an empty body.

Secondly, if the first difference is a modification to the code that changes both the condition and the body of an expression, only the change to the condition is retained. For example, changing the function parameters as well as the body will result in only the changed function parameters being retained. Changing the range and body of a for-loop means only the change to the range will be kept.

Lastly, if there are several changes to the body of an expression, only the first one is retained. For the full description of the heuristics, see appendix A, which shows JetBrains' description of the heuristics, as described in their supplementary materials (3) to their paper introducing the system (2).

III. Scientific article

Comparing the hint quality of a Small Language Model and a Large Language Model in automatic hint generation

Replacing the LLM inside the JetBrains Academy AI hint generation system with a RAG-augmented SLM

Casper Dekeling[1,3]
Supervisor(s): Gosia Migut[1]**, Anastasiia Birillo**[2]
[1]EEMCS, Delft University of Technology, The Netherlands
[2]JetBrains, Belgrade, Serbia
[3]JetBrains, Amsterdam, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Master of Computer Science
June 10, 2025

Name of the student: Casper Dekeling
Final project course: IN5000
Thesis committee: Marcus Specht[1], Arie van Deursen[1], Gosia Migut[1]

Abstract

The rapid advancement of Large Language Models (LLMs) in recent years is not without concerns, such as a lack of privacy, environmental impact, and financial concerns. It might therefore be beneficial to use Small Language Models (SLMs) instead, which are more accessible to be run by individuals or organisations, thus resulting in more control over the model. This research investigates whether we can replace an LLM with an SLM inside an AI hint-generation system, and achieve comparable hint quality, by conducting an expert study to validate generated hints based on a set of criteria and by conducting a student experiment, investigating student satisfaction and trust in the system. The expert results show that the hints generated by the SLM-powered system are slightly less personalised to the situation, are noticeably more misleading and more often suggest the wrong approach. The student experiment shows similar results for these criteria, and shows a slight decrease in the overall perceived helpfulness of the hints, trust in the system and willingness to continue using the system. The most prevalent complaint for the SLM-powered system was its inconsistency in the hint quality, as it generated good and useful hints in some contexts, but also suggested wrong and unusable hints too often. Thus, while replacing the LLM with an SLM has potential, as it is capable of generating useful hints, current SLMs are still too inconsistent.

Introduction

In education, it is vital that students stay engaged with the material when learning anything new [1]. To accomplish this, we must help students overcome obstacles that would demotivate them from continuing and finishing courses, such as being stuck on an assignment, or wanting further clarification on some material. More specifically, in programming education, students could face problems such as being unable to find a bug in their code, or not knowing what approach to take to solve a coding problem. [2]–[4]

The biggest problem in helping students with their issues is the need for enough teaching staff to be available for students to have the most effective learning environment [5]. This is an increasingly difficult challenge, given the growing ratio of students to teaching staff [6]. However, complementing the teaching staff with an AI tutor or hint system can help solve the shortage of staff [7]–[10], providing students with 24/7 access to help.

Additionally, having access to an AI tutor solves the problem where students might be uncomfortable seeking help from course staff [11], as the barrier is much lower to ask a non-human AI for help. Thus, we have a strong motivation to implement and optimise AI hint-generation systems such as CodeAid [9], the LLM Hint Factory [12], and the Help Tutor system [13]. JetBrains has also developed an AI hint system [14], which is the use case for this research. It is integrated into the IDE in their *JetBrains Academy* plugin. The system can give students hints based on their current implementation, guiding them to a solution to the problem without plainly giving them the answer.

Many AI hint-generation systems currently use an LLM like OpenAI's GPT-4 to generate code and text hints. However, this raises several concerns, especially in education, where large groups of students would use the system. Primarily, data privacy might be important for universities if they were to use the system. Not having full transparency and control over the students' data could discourage the use of AI hint-generation systems by universities. Additionally, financial and environmental concerns also play a role in the decision to use the system. Thus, it could be beneficial to replace the Large Language Model (LLM) with a Small Language Model (SLM), which a university or organisation could run locally and have full control over. This research will focus on whether an SLM, enhanced with Retrieval Augmented Generation (RAG), can generate hints of comparable quality to hints generated by an LLM.

Contributions and Research Questions

In this research, we investigate how the findings that SLMs can perform comparably to LLMs extend into a more niche domain: the Kotlin programming language. Being a much less prominent programming language, it will not be as prevalent in the training dataset used for an SLM as a programming language like Python would be. We investigate the difference in quality of the generated hints by conducting an expert study and determine student satisfaction with the system through a student experiment.

We balance the possible knowledge gap of an SLM by providing the model with a RAG module, and researching the difference RAG makes in the students' satisfaction with the system.

In this research, the SLM is part of a larger system that includes static analysis and processing of the generated hints [14], possibly further reducing any drawbacks that might arise from using an SLM instead of an LLM and potentially bringing the quality of hints generated by SLMs even closer to that of an LLM.

Thus, our contributions can be summarised as follows:

- Augmenting an SLM with RAG and evaluating it as part of a larger system that includes static analysis and processing of the generated hints.
- Evaluating the SLM-powered AI hint generation on a niche domain: the Kotlin programming language.
- Combining insights from an expert study with insights from a student experiment.

These contributions are encapsulated by answering the following research questions:

- How does replacing an LLM with an SLM as part of an AI hint-generation system affect the hint quality as determined by experts?
- How does replacing an LLM with an SLM as part of an AI hint-generation system affect student satifaction and trust in the system?

Related work

Incorporating AI tutors into a course to complement the teaching staff can help the students increase their learning and

task performance [6]. Multiple implementations of such an AI tutoring system have been developed, tested and deployed around the world in courses both in and outside the field of computer science. Since this research focuses on programming courses, this section does too, but a few examples of AI in education outside of computer science are still shown. We also discuss some concerns about integrating AI tutors into education. Lastly, this section also discusses Small Language Models (SLMs) and Retrieval Augmented Generation (RAG), a technique commonly used to enhance SLMs by providing them with access to more knowledge.

LLM tutoring systems in general education

A chatbot tutoring system powered by a RAG augmented LLM, developed and tested in collaboration with the Transilvania University of Brasov has shown promising preliminary results in their Virtual Instrumentation course [15]. It shows a potential to improve the students' learning experience by providing contextually accurate responses to their questions.

In [10], the authors compare the use of a custom RAG-augmented LLM tutor, a ChatGPT4 Turbo tutor, and not using any AI tutor in an introductory psychology course to see how they affect the students' learning. The students completed a writing assignment using either the RAG tutor, ChatGPT tutor, or no tutor and took a multiple-choice exam about the content of the assignment afterwards. The results show that students who had access to either of the two AI tutors scored significantly higher on the exam than students who did not have access to an AI tutor. In this study, there was no significant difference between the two AI tutors, but that might be due to the introductory nature of the course, diminishing the need for an external knowledge base. However, it clearly shows that AI tutors can be a valuable tool within education.

LLM tutoring systems in CS education

The latest LLM breakthroughs have been used to develop a system that provides students of an introductory computer science course with real-time feedback on their code style [16]. It is found that students who receive their feedback immediately are five times more likely to view it than students who receive the feedback with a delay. It is also found that students who viewed their feedback were more likely to apply it and make significant style changes to their code. However, there are also limitations to the system. There are still cases where the model responds with an inaccurate response or with a hallucination. Additionally, the model used a non-public and non-adjustable dataset, so it is quite rigid in its behaviour. Lastly, the model cannot provide high-quality feedback for some areas, such as indentation or major restructuring.

CodeAid is an LLM-powered programming assistant that was deployed in a programming course with 700 students, with the aim to give students helpful and technically correct responses without revealing solutions [9]. Students appreciated the constant availability of the assistant and its ability to clearly explain code or complex topics and assist in identifying errors. Students could ask the assistant lots of diverse questions without "having to talk to a human who will judge" them. They also appreciated the AI assistant over simply using a search engine because the AI assistant has more contextual knowledge, as it is hard to provide code to a search engine. Additionally, students could ask questions in the way they wanted to, instead of trying to match the title of the most relevant StackOverflow post. There were, however, also some drawbacks. The responses were often too brief and lacked depth, requiring students to ask follow-up questions to get a satisfactory answer. There were also concerns about the correctness of the responses, sometimes giving students misleading suggestions for fixing their code.

In [8], the authors also show that students reacted positively to an AI assistant, as they felt they had a personal tutor available at all times. The AI assistant also had built-in pedagogical guardrails and could be used to explain code snippets, improve code style and respond to questions on the course's discussion forum. The responses on the forum are not meant to replace human involvement, but rather complement it, as they are all subject to endorsement, amendment or deletion by staff members.

Dangers of using LLM tutors in education

While AI tutors can help students learn, there are also dangers and negative aspects we should be aware of [17]. There is an inherent distrust of the answers given by an AI tutor, and there are concerns about the quality of education and the value of a diploma. Since LLMs are at a point where they can solve any exercise from introductory programming courses [9], we need to implement guardrails ensuring the AI tutor will help the students solve the assignment themselves, instead of solving it for them. AI-generated hints can also cause frustration for students if the hint repeats things they already know and state the obvious [12].

Small Language Models (SLMs)

Small Language Models are compact versions of Large Language Models, designed to be more efficient at the same tasks. The models use fewer parameters and are trained on less data, making them more flexible and customisable. Additionally, the models are smaller, making it easier to run them locally, giving full control and improving the cost-efficiency, data privacy and environmental effects compared to an LLM that is run on a large supercomputer by a corporation [18]. However, the benefits of a smaller model are not without a cost, as the model's smaller size and reduced training set impact the quality of its output. Nevertheless, SLMs can still achieve comparable results to LLMs [19]–[23], especially when augmented with techniques like Retrieval Augmented Generation [18], [24] to cover gaps in the SLM's knowledge caused by the reduced training set, making SLMs a viable alternative to LLMs in certain contexts.

Retrieval Augmented Generation (RAG)

Retrieval Augmented Generation is a technique that allows a model to access additional information which was not nec-

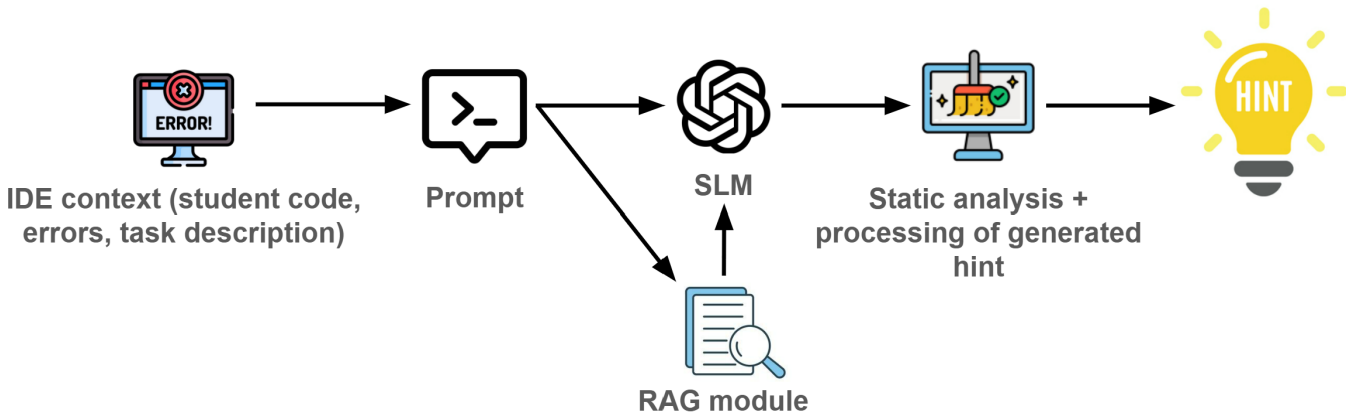

Figure 1: An overview of the AI hint-generation system.

essarily part of its training data. This means the model can generate responses based on more than just its parametric memory. Augmenting a model with RAG will change the way a prompt is handled. Instead of going straight to the model, the prompt is first processed by the RAG component. This component has access to a knowledgebase and will retrieve chunks of information from it based on their relevance to the request. The gathered information is then passed to the model, together with the original query, to generate a response that can use the relevant information from the knowledgebase. Additionally, the knowledgebase can easily be updated or replaced, resulting in much more flexibility [25]. Thus, the model can have access to the most up-to-date information without the need for constant fine-tuning or retraining [26]. RAG is thus especially useful in an educational context, since we can easily supply the model with a different knowledgebase for each course.

Using the benefits of RAG, it is possible to outperform state-of-the-art models on question-answering tasks by combining the strengths of "closed-book" (purely parametric) and "open-book" (retrieval-based) approaches [25]. It also achieves impressive results in other tasks such as Jeopardy question generation and fact verification. In general, the authors find the responses generated using RAG to be more factual, specific, and even more diverse. However, the authors also note that we should pay special attention to the knowledgebase we provide to a RAG model, as it can be a new source of bias or even incorrect information if the knowledgebase is not properly vetted.

The authors of [26] also implemented a framework to create RAG-enhanced AI tutors. They allow the user to upload their own knowledgebase, which will then be used to provide extra information to queries before they are passed to the LLM. The authors show that the system performs well on accuracy, relevance, and citation, meaning the responses were factually correct, aligned with the question and supported by citations from the relevant sources.

RAG can be used to reduce hallucinations in the responses provided by a model, resulting in high accuracy when the questions were within the intended scope [7]. Thus, RAG can be a key technique to enhance the factual correctness and reliability of the model [24].

Methodology

This section describes the methodology for implementing our SLM and RAG system, and describes the expert study and student experiment.

Implementing the SLM and RAG API

The current implementation of JetBrains' system uses their custom backend designed to communicate with several LLMs, like OpenAI's ChatGPT-4o. It does not support communication with any SLMs, so for this research, we implemented our own API using Ollama[1], allowing us to easily experiment with different SLMs. For the RAG component, we implemented the API using LlamaIndex[2]. An overview of the flow of our system when the student requests a hint can be seen in Figure 1

To find the optimal SLM and RAG configuration, we performed individual testing by comparing the quality of the hints and RAG outputs generated by the different systems for the same contexts. To account for the probabilistic behaviour of the models, we generated multiple hints per context for every model. The optimal configuration we found can be seen in Table 1

Generating expert study dataset

To test the quality of the SLM-generated hints compared to the LLM-generated hints for the same context, we need a diverse dataset of hint situations. To obtain this dataset, we

[1]Ollama. Retrieved from `https://ollama.com`

[2]LlamaIndex. Retrieved from `https://docs.llamaindex.ai/en/stable/`

[3]Ollama: Qwen2.5-Coder:7b, a version of the Qwen2.5 model fine-tuned for code generation, code reasoning, and code fixing. Retrieved from `https://ollama.com/library/qwen2.5-coder:7b`

[4]ChromaDB. Retrieved from `https://docs.trychroma.com/docs/overview/introduction`

[5]BAAI/bge-base-en-v1.5. Retrieved from `https://huggingface.co/BAAI/bge-base-en-v1.5`

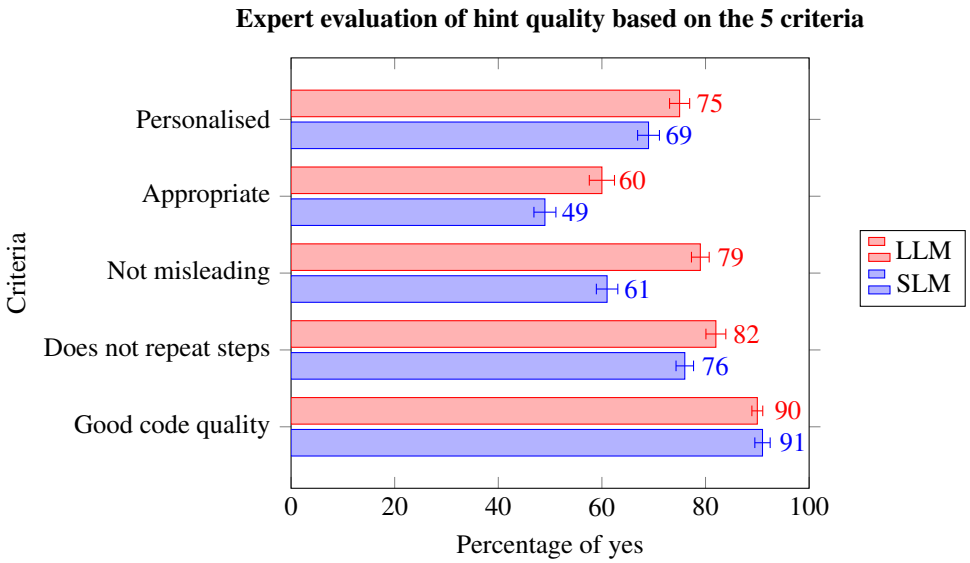

Figure 2: The average percentage of experts that answered yes on each of the five criteria. Note the negative phrasing of some criteria, which was done to make a higher percentage correspond to higher hint quality. For example, the experts were asked whether a hint was **misleading**, but in the figure, these results are inverted and instead presented as **not misleading**, to align all criteria and ensure a higher score is positive for all criteria. The error bars represent the average standard error over the data.

SLM Model	`Qwen2.5-Coder:7b`[3]
RAG knowledgebase	Kotlin documentation
RAG vector storage	Chroma Database[4]
RAG embedding model	the Beijing Academy of Artificial Intelligence's `bge-base-en-v1.5`[5]
RAG similarity measure	Cosine distance

Table 1: The optimal configuration of the SLM-RAG used for the AI Hint Generation system, as found from individual testing.

use the *Kotlin Onboarding: Introduction* course from the JetBrains Academy plugin. The course consists of projects, such as implementing games like Hangman or Mastermind. Each project is divided into small tasks, guiding the student in implementing the project.

From the course, we use 5 projects and choose 2 tasks for each project. We select the tasks in such a way that we have a varied set of tasks; a balance of simple and more complex tasks, and different types of tasks, such as user input/output, full game logic and implementing helper functions. Within the tasks, we also generate multiple contexts per task, corresponding to multiple stages of a student's implementation, such as a student not knowing how to start, being stuck halfway through their implementation, or being almost finished but having a bug in their code. Lastly, to account for probabilistic outputs, we also generate multiple hints for both the SLM and LLM for every context. If the generations are significantly different, either in the text hint or the code hint, we include all of them in our dataset. In total, the dataset consists of 59 generated hints.

Expert validation criteria

To assess the quality of the generated hints, we ask experts to rate each generated hint using a set of validation criteria adapted from [14] and [27]. Specifically, the criteria used are as follows.

- **Personalised** - Does the hint refer to the student's code, and is it a logical extension of the student's code?
- **Appropriate** - Is the hint a suitable next step to solve the task, given the current state of the student's code and desired outcome?
- **Misleading information** - Does the hint contain any misleading information or hallucinations? e.g. using incorrect functions or non-existent variables.
- **Intersection** - Is the hint already fully or partially implemented in the student's code? i.e. does the hint repeat steps the student has already done?
- **Code quality** - is the code compilable, does it not have common code quality violations, such as incorrect brackets and does it conform to code conventions?
- **Level of detail** - is the hint a high-level description or a specific bottom-out hint?

All of the criteria are binary criteria, with all criteria except for the *level of detail* being yes/no criteria. The set of criteria was chosen because they cover the essential elements that define a good hint, and thus the most important areas where the SLM should perform similarly to the LLM. Additionally, criteria like length of the code hint, as used in [14] are covered by the static analysis component of the system, and thus

can be excluded from our research, as this is not the part of the system we are evaluating.

Expert study setup

To qualify as an expert for the study, both programming knowledge and teaching experience is required. Specifically, our group of experts is a mix of computer science professors and teaching assistants from the Delft University of Technology. We executed the study with 6 professors and 10 teaching assistants, dividing the dataset into three subsets for the professors and two subsets for the teaching assistants and having each expert evaluate one subset. Thus, every data point was evaluated by 2 professors and 5 teaching assistants.

The experts filled in a survey, where for each scenario, the student context is first shown and explained: what task the student is trying to implement and what mistake the student made or what the student is struggling with. Then, experts are shown all the hints generated for that context by both systems, with the text and code hint of each generation grouped together. The hints are rated blindly, meaning the expert has no bias induced by knowing what system they are evaluating.

Before being sent to the experts, the survey was subjected to an iterative feedback loop by first performing the study in a testing scenario with peers. This ensures good quality and makes it as easy and clear as possible, thus minimising any noise in the data caused by misunderstanding the survey or the criteria. The feedback resulted in improvements such as better formatting and clearer explanations for certain parts of the survey.

Student experiment setup

After experts have evaluated the quality of the hints generated by the system, we perform a student experiment, emulating the system's real use case. Two groups of students implement one of the projects from the *Kotlin Onboarding: Introduction* course, while using the hint-generation system to ask for hints if they are stuck. One group will only have access to the original system, using an LLM, while the other group will only have access to the RAG-augmented SLM system. After the students have finished the project, they will complete a satisfaction survey, where they will rate their experience with the system and their trust in the hints.

Each group consists of 14 students, with a comparable distribution of English proficiency, study progress, general coding experience, Kotlin experience, and experience with using AI systems.

The survey is adapted from [14], asking students about their satisfaction with the system, such as whether they found the hints helpful, what type of hints they preferred, and whether they would continue using the system. Additionally, the survey includes the same criteria from the expert study, now as a 5-point Likert scale, where students rate all hints as a whole, instead of rating each hint on its own. Lastly, the survey also gathers general information about the students to see whether these characteristics influence the students' satisfaction and to ensure comparable demographics in each group.

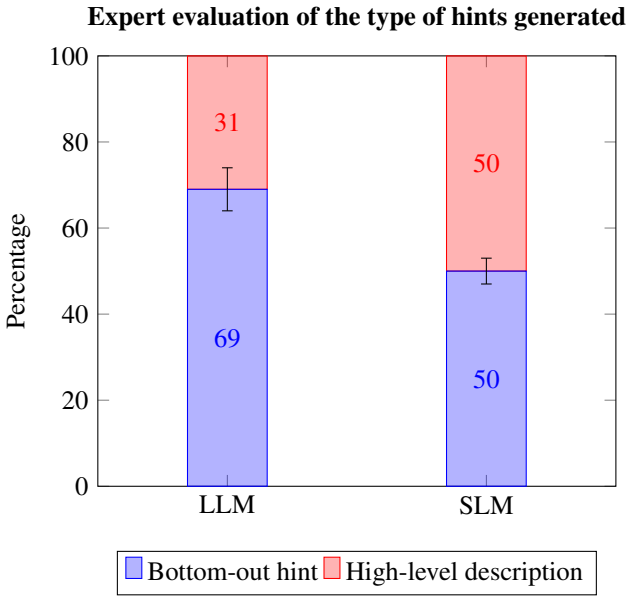

Figure 3: The average percentage of experts that classified each generated hint as a concrete, bottom-out hint, versus a more abstract, high-level description. The error bars represent the average standard error over the data.

The characteristics include their English proficiency, study progress, experience with programming and Kotlin, and their opinion on AI and its usefulness.

Again, the experiment is tested iteratively, by first performing it with peers and students to gather feedback, before performing the experiment with the actual group of students. The individual round showed that parts of the experiment instructions and task descriptions in the course needed rephrasing. Additionally, it highlighted some problems that could occur when setting up the environment. Thus, the commonly encountered problems and their solutions were also included in the instructions.

Results

Expert study

The results from the expert study can be seen in Figures 2 and 3. We see from Figure 2 that the *code quality* of hints generated by the SLM is of a similar standard as the hints generated by the LLM. The SLM scores slightly worse on the *personalised* and *intersection* criteria. Thus, on average, the hints generated by the SLM are less of a logical extension of the student's code and more often repeat steps that the student has already implemented. Lastly, on average, we see that the SLM-generated hints are significantly less *appropriate*, and more *misleading*. This means the hints more often suggest a suboptimal next step, or possibly even contain an incorrect next step, with functions being used incorrectly or containing erroneous code.

From Figure 3, we see that the LLM generates hints that are classified as 69% bottom-out hints, and 31% as high-level descriptions on average, meaning the LLM more often gener-

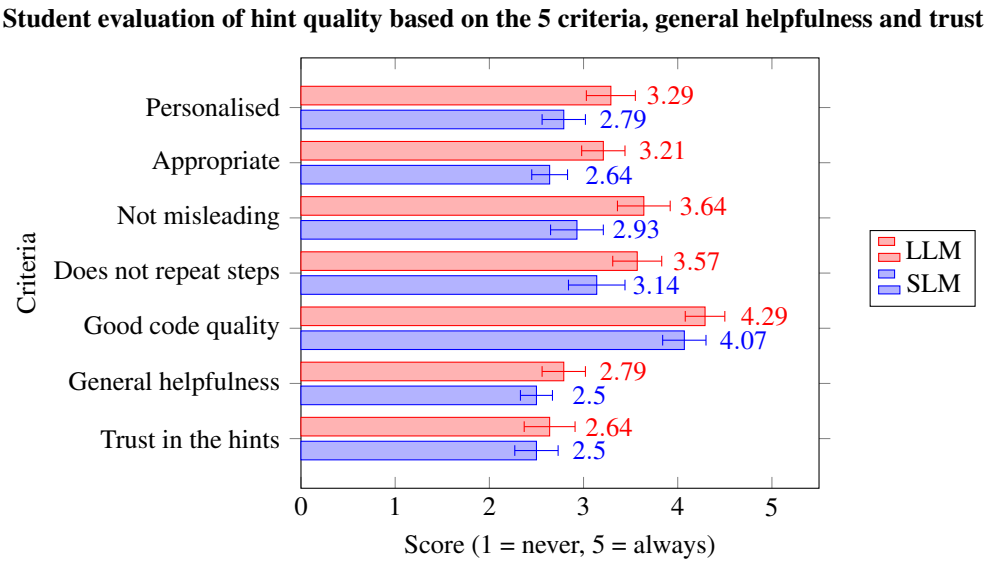

Figure 4: The average score students gave on the same five criteria that the experts evaluated and their opinion on the helpfulness and trust in the hints, using a five-point Likert-scale, with 5 being the highest score, and 1 the lowest. Note the negative phrasing of some criteria, which was done to make a higher percentage correspond to higher hint quality. For example, the students were asked whether hints were ever **misleading**, but in the figure, these results are inverted and instead presented as **not misleading**, to align all criteria and ensure a higher score is positive for all criteria. The error bars represent the average standard error over the data.

ates hints that plainly tell the student what to do. The SLM, on the other hand, generates hint that are on average classified as 50% high-level descriptions, and 50% as bottom-out hints. Thus, on average, half of the hints generated by the SLM give a more abstract push in the right direction, and the other half are concrete bottom-out hints.

To assess the reliability of the results gathered from different experts, we computed Krippendorff's Alpha [28]. We obtained a Krippendorff's Alpha of 0.705, suggesting moderate agreement and consistency among the experts. While not perfect, this level of agreement shows that the experts were reasonably aligned in their ratings, but there is still room for improvement in terms of consistency.

Student experiment

The results from the student experiment can be seen in Figures 4 and 5. Figure 4 shows the students' opinion on the same five criteria as used in the expert study, except as a 5-point Likert scale instead of binary criteria. It also shows how helpful the students found the hints, and how much trust they have in the hints and the system.

We see that the SLM's scores for the five criteria are between 10-20% worse than the LLM's scores, except for the code quality, where it is only 5% worse. Similar to the expert study, we see the biggest performance drop in the appropriateness of the hint and how often the hint contains misleading information.

For the general helpfulness and trust in the hints, we also see a slight decrease for the SLM, seeing a 10% drop in helpfulness, and a 5% drop in trust.

In Figure 5, we see that 69% of the students who used the LLM system would continue using hints, though just over half of these students would only continue if certain changes are made to the system. The two changes that students would like to see are the ability to give some form of feedback or ask a return question to the AI, giving the student the ability to give feedback on the hint and directly generate a new hint using this feedback. Additionally, the hints would need to generate faster and be more consistent, as even the LLM occasionally generates hints that are not useful at all.

We see that 64% of the students who used the SLM system would continue using hints. However, here, more than three-quarters of these students would need certain changes in the system. Here, the main concern is the consistency of the correctness and usefulness of the hints. The system does generate correct and usable hints, but it also generates too many incorrect and unusable hints.

We also collected open feedback for the experiment, which showed that, in general, the hints are very hit or miss. If the AI can detect the mistake or problem in the code, the generated hint is generally of high quality and useful. However, if it does not see the problem, the hint is often unusable, suggesting something completely irrelevant, like removing a previously implemented function that has nothing to do with the current task, or removing some newlines. This holds for both the LLM and SLM systems, but was more prevalent in the SLM system.

Another common improvement that students would like to see is the possibility to communicate with the AI. Students would like to be able to tell the model what they are struggling with, ask it questions, or give feedback on the generated hints and generate a new hint taking this feedback into account. Even though this is not relevant to the research questions, this

feedback is mentioned because it occurred often.

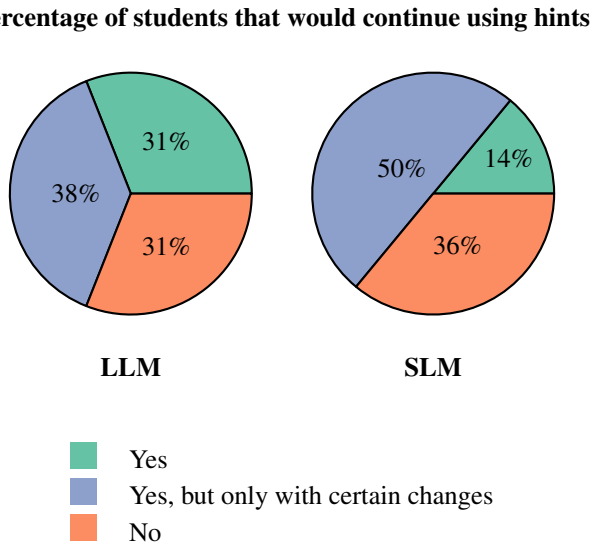

Figure 5: Percentage of students that would continue using the AI Hint Generation system. The changes students would want, to continue using the system were given as open answers.

Discussion

From the expert study, we can see that the main quality decrease of the SLM-generated hints is the appropriateness and correctness of the hints, meaning there is often a better next step towards a solution than the one suggested in the hint. Additionally, the hint might not always be correct and can contain mistakes, such as calling a helper function that does not exist or using the wrong library function.

The appropriateness of the LLM-generated hints was already relatively low, and the decrease for the SLM means that more than half of the hints suggest a suboptimal next step, on average. Together with the increase in misleading hints, this becomes a concern for the use of an SLM inside the hint-generation system, as the wrong next step can cause the student to have more trouble with the assignment, and misleading information can teach the student incorrect information or practices. One mistake that was seen often was suggesting using a helper function implemented in the author's solution, without actually implementing it. The signature of the author's helper functions are passed to the model to give an idea of what helper functions could be implemented by the student, but instead are sometimes interpreted by the model as being implemented already.

Similar results are seen in the student experiment, where the five criteria are also evaluated, and we see a similar drop in ratings, with the appropriateness of the hints and the amount of misleading information in the hints being responsible for the biggest decrease in quality. For the general helpfulness of the hints and trust in the system, the students rated both the LLM and SLM systems around the range of a 2.5 to 3.0 score on a 5-point Likert scale. With the margins of error, these are quite close together, with the LLM system scoring slightly better in both categories.

We also see that 69% of the students who used the LLM system would continue using it, and 64% of the students who used the SLM system would too. However, for the SLM system, more than three quarters of the students who would continue using the system say this is only if certain changes were implemented in the system, mostly focusing on more consistent hints. For the LLM, almost half of the students would continue using the system as it is.

Thus, we see that using an SLM shows promise, but it is not good enough yet. Students express that they would like to keep using the system, even when powered by an SLM, but the consistency is still lacking compared to the LLM. With the field constantly developing, this problem could be solved as SLMs get better over time. Alternatively, using larger SLMs could mitigate this problem, though this reduces the accessibility of the system, as larger SLMs require more computational power to run.

Limitations

One limitation of this research is the reliability of the expert study. We obtained a Krippendorff's Alpha of 0.705, suggesting moderate agreement and consistency between the experts. Ideally, it should be 0.800 or higher to indicate the most reliable results.

Another limitation is the length of the student experiment. The experiment was a short session where students completed one small programming project with help from the AI hint-generation system. However, we could gain more insights from a study where students use the system for a prolonged period of time, spanning multiple sessions and assignments.

Conclusions and Future Work

In this research, we see that replacing the LLM inside an AI hint-generation system with an SLM results in a noticeable drop in hint quality as determined by experts. The appropriateness of the hints is responsible for the biggest decrease in quality, together with an increase in misleading information in the hints.

From the student experiment, we see that the students perceive a similar drop in hint quality, with the appropriateness and misleading information again resulting in the biggest decrease in quality. However, we see that the students' rating of the general helpfulness of the hints and trust in the system is only slightly worse for the SLM, compared to the LLM. Additionally, for both systems, we observe a similar percentage of students who express that they would continue using this system. However, for the SLM, a larger portion of these students would only do so if certain improvements were made, such as more consistent hints.

Thus, while there is potential in replacing the LLM inside a hint-generation system with an SLM, the drop in hint quality is still noticeable, both by experts and students. The SLM can generate high-quality, usable hints, but it is too inconsistent in understanding the problem that the student is strug-

gling with. This may be mitigated by using newer SLMs that are continuously being developed and improved, or by using a larger SLM. This could be the focus of future research, together with a more extensive student experiment.

References

[1] G. Kanaparan, R. Cullen, and D. Mason, "Self-Efficacy and Engagement as Predictors of Student Programming Performance," vol. 282, Jun. 2013. [Online]. Available: https://aisel.aisnet.org/pacis2013/282.

[2] J. Whalley, A. Settle, and A. Luxton-Reilly, "A Think-Aloud Study of Novice Debugging," *ACM Trans. Comput. Educ.*, vol. 23, no. 2, Jun. 2023, Place: New York, NY, USA Publisher: Association for Computing Machinery. DOI: 10.1145/3589004. [Online]. Available: https://doi.org/10.1145/3589004.

[3] J. Prather, R. Pettit, K. H. McMurry, *et al.*, "On Novices' Interaction with Compiler Error Messages: A Human Factors Approach," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17, event-place: Tacoma, Washington, USA, New York, NY, USA: Association for Computing Machinery, 2017, pp. 74–82, ISBN: 978-1-4503-4968-0. DOI: 10.1145/3105726.3106169. [Online]. Available: https://doi.org/10.1145/3105726.3106169.

[4] R. Smith and S. Rixner, "The Error Landscape: Characterizing the Mistakes of Novice Programmers," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19, event-place: Minneapolis, MN, USA, New York, NY, USA: Association for Computing Machinery, 2019, pp. 538–544, ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287394. [Online]. Available: https://doi.org/10.1145/3287324.3287394.

[5] J. Hattie, "The applicability of Visible Learning to higher education.," *Scholarship of Teaching and Learning in Psychology*, vol. 1, no. 1, pp. 79–91, 2015. DOI: https://doi.org/10.1037/stl0000021. [Online]. Available: https://doi.org/10.1037/stl0000021.

[6] A. Maedche, C. Legner, A. Benlian, *et al.*, "AI-Based Digital Assistants," *Business & Information Systems Engineering*, vol. 61, no. 4, pp. 535–544, Aug. 2019, ISSN: 1867-0202. DOI: 10.1007/s12599-019-00600-8. [Online]. Available: https://doi.org/10.1007/s12599-019-00600-8.

[7] I. Ma, A. K. Martins, and C. V. Lopes, *Integrating AI Tutors in a Programming Course*, _eprint: 2407.15718, 2024. [Online]. Available: https://arxiv.org/abs/2407.15718.

[8] R. Liu, C. Zenke, C. Liu, A. Holmes, P. Thornton, and D. J. Malan, "Teaching CS50 with AI: Leveraging Generative Artificial Intelligence in Computer Science Education," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2024, event-place: Portland, OR, USA, New York, NY, USA: Association for Computing Machinery, 2024, pp. 750–756, ISBN: 9798400704239. DOI: 10.1145/3626252.3630938. [Online]. Available: https://doi.org/10.1145/3626252.3630938.

[9] M. Kazemitabaar, R. Ye, X. Wang, *et al.*, "CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24, event-place: Honolulu, HI, USA, New York, NY, USA: Association for Computing Machinery, 2024, ISBN: 9798400703300. DOI: 10.1145/3613904.3642773. [Online]. Available: https://doi.org/10.1145/3613904.3642773.

[10] J. J. Slade, A. Hyk, and R. A. R. Gurung, "Transforming Learning: Assessing the Efficacy of a Retrieval-Augmented Generation System as a Tutor for Introductory Psychology," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 0, no. 0, p. 10 711 813 241 275 509, _eprint: https://doi.org/10.1177/10711813241275509. DOI: 10.1177/10711813241275509. [Online]. Available: https://doi.org/10.1177/10711813241275509.

[11] A. Ryan and P. Pintrich, ""Should I ask for help?" The role of motivation and attitudes in adolescents' help seeking in math class.," *Journal of Educational Psychology*, vol. 89, no. 2, pp. 329–341, 1997. DOI: https://doi.org/10.1037/0022-0663.89.2.329. [Online]. Available: https://psycnet.apa.org/doi/10.1037/0022-0663.89.2.329.

[12] R. Xiao, X. Hou, and J. Stamper, "Exploring How Multiple Levels of GPT-Generated Programming Hints Support or Disappoint Novices," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24, ACM, May 2024. DOI: 10.1145/3613905.3650937. [Online]. Available: http://dx.doi.org/10.1145/3613905.3650937.

[13] V. Aleven, I. Roll, B. M. McLaren, and K. R. Koedinger, "Help Helps, But Only So Much: Research on Help Seeking with Intelligent Tutoring Systems," *International Journal of Artificial Intelligence in Education*, vol. 26, no. 1, pp. 205–223, Mar. 2016, ISSN: 1560-4306. DOI: 10.1007/s40593-015-0089-1. [Online]. Available: https://doi.org/10.1007/s40593-015-0089-1.

[14] A. Birillo, E. Artser, A. Potriasaeva, *et al.*, "One Step at a Time: Combining LLMs and Static Analysis to Generate Next-Step Hints for Programming Tasks," in *Proceedings of the 24th Koli Calling International Conference on Computing Education Re-*

search, ser. Koli Calling '24, New York, NY, USA: Association for Computing Machinery, 2024, ISBN: 979-8-4007-1038-4. DOI: 10.1145/3699538.3699556. [Online]. Available: https://doi.org/10.1145/3699538.3699556.

[15] H. Modran, I. C. Bogdan, D. Ursuțiu, C. Samoila, and P. L. Modran, "LLM Intelligent Agent Tutoring in Higher Education Courses using a RAG Approach," *Preprints*, Jul. 2024, Publisher: Preprints. DOI: 10.20944/preprints202407.0519.v1. [Online]. Available: https://doi.org/10.20944/preprints202407.0519.v1.

[16] J. Woodrow, A. Malik, and C. Piech, "AI Teaches the Art of Elegant Coding: Timely, Fair, and Helpful Style Feedback in a Global Course," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2024, event-place: Portland, OR, USA, New York, NY, USA: Association for Computing Machinery, 2024, pp. 1442–1448, ISBN: 9798400704239. DOI: 10.1145/3626252.3630773. [Online]. Available: https://doi.org/10.1145/3626252.3630773.

[17] H. Carbonel and J.-M. Jullien, "Emerging tensions around learning with LLM-based chatbots: A CHAT approach," *Proceedings of the International Conference on Networked Learning*, vol. 14, no. 1, Apr. 2024. DOI: 10.54337/nlc.v14i1.8084. [Online]. Available: https://journals.aau.dk/index.php/nlc/article/view/8084.

[18] F. Wang, Z. Zhang, X. Zhang, *et al.*, *A Comprehensive Survey of Small Language Models in the Era of Large Language Models: Techniques, Enhancements, Applications, Collaboration with LLMs, and Trustworthiness*, _eprint: 2411.03350, 2024. [Online]. Available: https://arxiv.org/abs/2411.03350.

[19] N. Kotalwar, A. Gotovos, and A. Singla, *Hints-In-Browser: Benchmarking Language Models for Programming Feedback Generation*, _eprint: 2406.05053, 2024. [Online]. Available: https://arxiv.org/abs/2406.05053.

[20] G. Team, M. Riviere, S. Pathak, *et al.*, *Gemma 2: Improving Open Language Models at a Practical Size*, _eprint: 2408.00118, 2024. [Online]. Available: https://arxiv.org/abs/2408.00118.

[21] Qwen, : A. Yang, *et al.*, *Qwen2.5 Technical Report*, _eprint: 2412.15115, 2025. [Online]. Available: https://arxiv.org/abs/2412.15115.

[22] S. Hu, Y. Tu, X. Han, *et al.*, *MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies*, _eprint: 2404.06395, 2024. [Online]. Available: https://arxiv.org/abs/2404.06395.

[23] M. Abdin, J. Aneja, H. Awadalla, *et al.*, *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*, _eprint: 2404.14219, 2024. [Online]. Available: https://arxiv.org/abs/2404.14219.

[24] S. Liu, Z. Yu, F. Huang, Y. Bulbulia, A. Bergen, and M. Liut, "Can Small Language Models With Retrieval-Augmented Generation Replace Large Language Models When Learning Computer Science?" In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE 2024, event-place: Milan, Italy, New York, NY, USA: Association for Computing Machinery, 2024, pp. 388–393, ISBN: 979-8-4007-0600-4. DOI: 10.1145/3649217.3653554. [Online]. Available: https://doi.org/10.1145/3649217.3653554.

[25] P. Lewis, E. Perez, A. Piktus, *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

[26] C. Dong, *How to Build an AI Tutor that Can Adapt to Any Course and Provide Accurate Answers Using Large Language Model and Retrieval-Augmented Generation*, _eprint: 2311.17696, 2024. [Online]. Available: https://arxiv.org/abs/2311.17696.

[27] L. Roest, H. Keuning, and J. Jeuring, "Next-Step Hint Generation for Introductory Programming Using Large Language Models," in *Proceedings of the 26th Australasian Computing Education Conference*, ser. ACE '24, event-place: Sydney, NSW, Australia, New York, NY, USA: Association for Computing Machinery, 2024, pp. 144–153, ISBN: 9798400716195. DOI: 10.1145/3636243.3636259. [Online]. Available: https://doi.org/10.1145/3636243.3636259.

[28] K. Krippendorff, *Computing Krippendorff's alpha-reliability*, 2011. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=de8e2c7b7992028cf035f8d907635de871ed627d.

References

Abbasi Yadkori, Y., Kuzborskij, I., György, A., and Szepesvari, C. To believe or not to believe your llm: Iterative prompting for estimating epistemic uncertainty. *Advances in Neural Information Processing Systems 37*, 58077–58117.

Birillo, A., Artser, E., Potriasaeva, A., Vlasov, I., Dzialets, K., Golubev, Y., Gerasimov, I., Keuning, H., and Bryksin, T. One Step at a Time: Combining LLMs and Static Analysis to Generate Next-Step Hints for Programming Tasks. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2024), Koli Calling '24, Association for Computing Machinery.

Birillo, A., Artser, E., Potriasaeva, A., Vlasov, I., Dzialets, K., Golubev, Y., Gerasimov, I., Keuning, H., and Bryksin, T. Supplementary materials for the paper "One Step at a Time: Combining LLMs and Static Analysis to Generate Next-Step Hints for Programming Tasks", Sept. 2024.

Ding, Y., and Shi, T. Sustainable LLM Serving: Environmental Implications, Challenges, and Opportunities : Invited Paper. In *2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC)* (2024), pp. 37–38.

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. Exploring the Limits of Language Modeling, 2016. _eprint: 1602.02410.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kazemitabaar, M., Ye, R., Wang, X., Henley, A. Z., Denny, P., Craig, M., and Grossman, T. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2024), CHI '24, Association for Computing Machinery. event-place: Honolulu, HI, USA.

Liu, R., Zenke, C., Liu, C., Holmes, A., Thornton, P., and Malan, D. J. Teaching CS50 with AI: Leveraging Generative Artificial Intelligence in Computer Science Education. In *Proceedings of the 55th ACM Technical Symposium on Computer Science*

Education V. 1 (New York, NY, USA, 2024), SIGCSE 2024, Association for Computing Machinery, pp. 750–756. event-place: Portland, OR, USA.

Ma, I., Martins, A. K., and Lopes, C. V. Integrating AI Tutors in a Programming Course, 2024. _eprint: 2407.15718.

Maedche, A., Legner, C., Benlian, A., Berger, B., Gimpel, H., Hess, T., Hinz, O., Morana, S., and Söllner, M. AI-Based Digital Assistants. *Business & Information Systems Engineering 61*, 4 (Aug. 2019), 535–544.

Ryan, A., and Pintrich, P. "Should I ask for help?" The role of motivation and attitudes in adolescents' help seeking in math class. *Journal of Educational Psychology 89*, 2 (1997), 329–341.

Slade, J. J., Hyk, A., and Gurung, R. A. R. Transforming Learning: Assessing the Efficacy of a Retrieval-Augmented Generation System as a Tutor for Introductory Psychology. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting 0*, 0, 10711813241275509. _eprint: https://doi.org/10.1177/10711813241275509.

Appendix

A. Heuristics for static analysis

Heuristic	Description	Application Examples
Additive Statement Isolation	If the first difference is an addition, only the statement is retained	Function → Replace body with TODO("Not yet implemented") While, For → Remove body If → Remove body and else block When → Remove entries Return → Proceed to simplify the expression following the return statement
Intrinsic Structure Modification Focus	If the first difference is a modification that involves changes to the condition and the body of an expression, only the change related to the condition remains.	Function → Change the function parameters While → Change the condition For → Change the loop range or the loop parameter If → Change the condition or else block When → Change the subject expression
Internal Body Change Detection	If there are several changes to the body of an expression, we only retain the first one.	Function, While, For, If, When, Return → Change the first difference in its body

Table A1: Heuristics used in the static analysis step of the JetBrains AI Hint generation system as described in (3)
. The heuristics control the size of the generated code-hint to ensure it corresponds to one logical step.

B. IDE view of the hint generation system

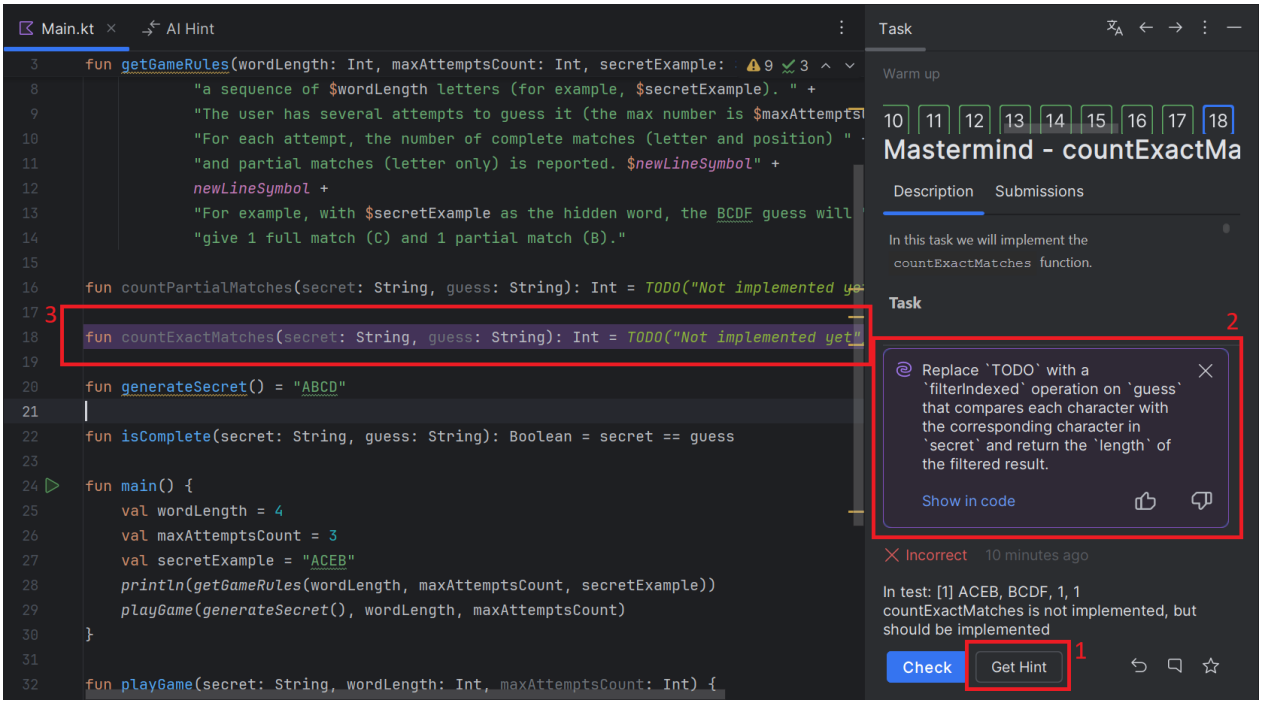

Figure A1: What the student sees while using the JetBrains Academy plugin after pressing the 'Get Hint' button, highlighted in red at number 1. At number 2, the generated text hint is shown, and at number 3, the relevant line inside the code is highlighted.

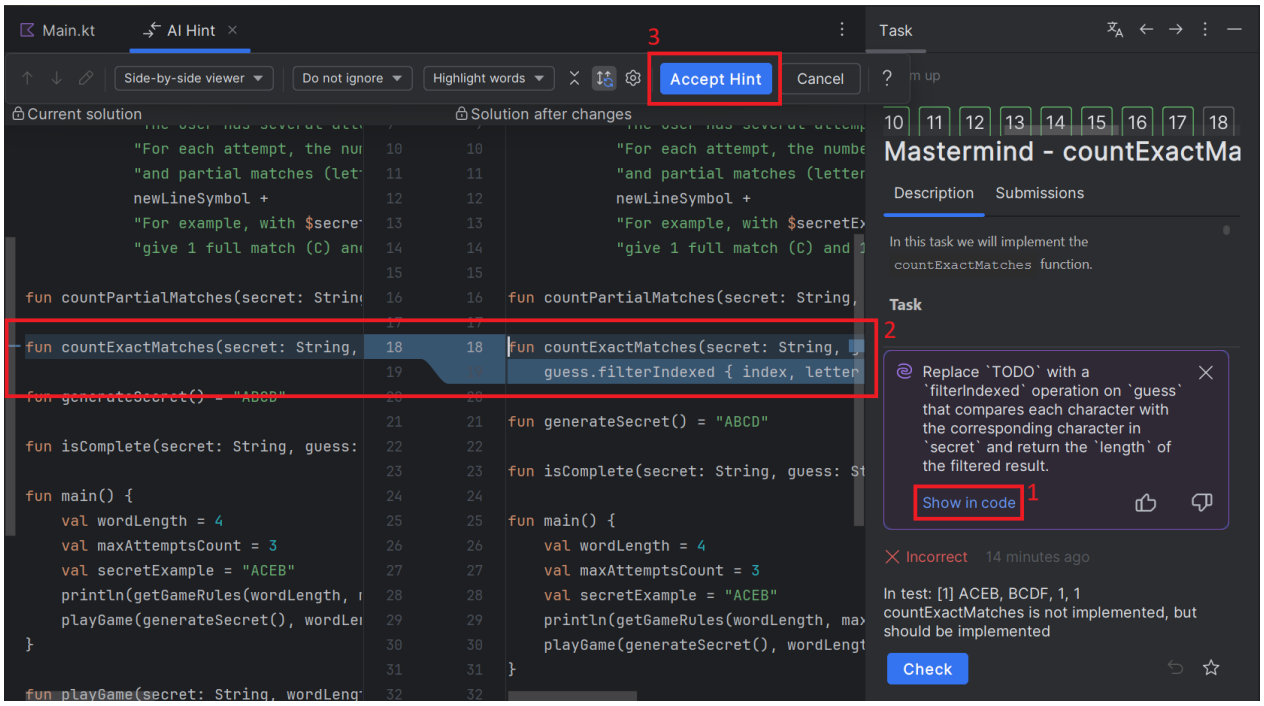

Figure A2: What the student sees while using the JetBrains Academy plugin after pressing the 'Show in code' (highlighted in red at number 1) button after receiving the text hint. At number 2, the generated code hint is shown compared to the student's current code, and at number 3, the student can press the 'Accept Hint' button to incorporate the suggested change into their code.