Study of Performance Improvement Techniques for Code Generation in Large Language Models

Swapna Shingade[1], Pratik Bhilore[2], Juie Pachupate[2], Sumeet Gaikwad[2], Akash Pandit[2]

[1]Assistant Professor, Department of Artificial Intelligence and Data Science, PVGCOET, Maharashtra, India.
[2]Student, PVGCOET, Maharashtra, India

Abstract - *Advancements in AI and Machine Learning have led to the creation of Large Language Models (LLMs) capable of generating human-like text and code. Despite this, generating accurate and efficient code remains a challenge. The study surveys performance improvement techniques for LLM code generation, focusing on Fine-Tuning, Prompt Design, and Context Awareness.*

Fine-tuning enhances LLM by adjusting the models on specific coding datasets, improving adaptability to different coding tasks. Prompt Design involves clear crafting and precise prompts to guide LLMs, enhancing the quality and accuracy of generated code. Context Awareness equips LLM with the ability to maintain and utilize context, ensuring coherence and consistency in code generation.

Through empirical analysis and case studies, the survey evaluates the techniques' impact on code generation performance. The findings aim to provide guidelines for optimizing LLMs, contribute to more reliable and efficient code generation, and improve software development processes.

Key Words: **Large Language Models (LLMs), Fine-Tuning, Prompt Designing, Context awareness, Code Generation**

1. INTRODUCTION

The rapid advancements in Artificial Intelligence and Machine Learning have led to the development of Large Language Models (LLMs) capable of generating human-like text. Among their many applications, code generation has emerged as a particularly promising area, offering the potential to significantly enhance software development processes. However, the performance of these models in generating accurate and efficient code remains a critical challenge. The survey paper aims to explore different performance improvement techniques for code generation in LLM. By examining the latest research and methodologies, the survey seeks to identify strategies that can enhance the accuracy, efficiency, and overall effectiveness of code generation models.

The paper dives into three main techniques: Fine-tuning, Prompt Design, and Context Awareness. Fine-tuning involves adapting pre-trained models to specific tasks or domains, thereby improving their performance on those tasks. Prompt Design focuses on crafting effective prompts that guide the model to generate more relevant and accurate code. Context Awareness emphasizes the importance of incorporating both in-file and cross-file contexts to provide the model with a comprehensive understanding of the codebase. By understanding the effects of these techniques on the quality of code generation, the study provides a comprehensive overview of the current state of the art, highlighting key techniques and their impact on the performance of LLMs in code generation tasks.

2. INTRODUCTION TO FINE-TUNING:

Fine-tuning in large language models (LLMs) involves the process of adapting a pre-trained model by optimizing its parameters on a task-specific, smaller dataset. The procedure leverages the pre-existing knowledge encoded in the model, to allow specialization in peculiar tasks, such as text classification, translation, or sentiment analysis. During fine-tuning, the model's weights are adjusted through gradient descent, refining its ability to perform the new task while retaining general language understanding. It typically requires balancing between preserving the model's generalization capabilities and adapting it to the specific characteristics of the new data, ensuring optimal performance on the target task without overfitting.

Retraining an LLM on task-specific data not only increases its ability to encode contextual knowledge but also significantly improves its performance in generating more relevant content. The minimal human-labeled seed tasks are used to generate more data. In this way, it is ensured that the quality and relevance of the tasks in the dataset are maintained while the manual effort is minimized. Over time, various traditional fine-tuning approaches have been utilized, including supervised fine-tuning on labeled datasets, transfer learning where pre-trained models are adapted to new tasks, and domain adaptation techniques aimed at enhancing model performance in specific contexts. In recent advances, full fine-tuning and various parameter-efficient fine-tuning methods have been proposed.

2.1. Full fine-tuning:

Full fine-tuning involves adjusting all the parameters of a pre-trained model to adapt it to a new task or dataset. The method retrains the entire model, allowing for extensive modifications to optimize its performance for specific applications. Even though full fine-tuning can achieve high levels of accuracy and task relevance, it demands considerable computational resources and memory, making it a more resource-intensive process compared to parameter-efficient techniques.

2.2. Parameter-efficient fine-tuning:

Parameter-efficient fine-tuning (PEFT)[13] refers to techniques that adapt pre-trained models to new tasks by updating only a small subset of parameters or introducing additional structures, thereby reducing computational and memory requirements while maintaining the model's core capabilities.

The method significantly enhances the base model's performance without demanding extensive computational power or large training datasets. The effectiveness of parameter-efficient fine-tuning (PEFT) techniques has indeed been well-established in prior works for small language models, demonstrating their superiority over full fine-tuning[13]. This promising foundation opens exciting opportunities for further exploration of PEFT in large language models (LLMs) (≥1B parameters)[13], which is expected to yield valuable insights and advancements in the field[13].

The study examined the application of parameter-efficient fine-tuning (PEFT) techniques to large language models (LLMs), focusing on how these methods can be adapted to enhance performance and efficiency in code generation. The study incorporates five parameter-efficient fine tuning (PEFT) techniques namely – Low-Rank Adaptation(LoRA), IA3(Input-Aware Adaptation with Attention), Prompt tuning, Prefix tuning and Quantized Low-Rank Adaptation (QLoRA)[13].

2.2.1. LoRA (Low-Rank Adaptation):

The technique consists of freezing the model weights and injecting low-rank trainable matrices into the attention layers of the transformer architecture[13], thereby drastically reducing the number of trainable parameters. Rather than updating all model parameters, a set of low-rank matrices is introduced and trained while the majority of the original model parameters are kept static. While allowing for task-specific adjustments with reduced computational overhead focusing on the adaptation of these low-rank matrices, the core functionality of the pre-trained model is preserved.

2.2.2 IA3 (Input-Aware Adaptation):

IA3 (Input-Aware Adaptation with Attention)[13] is an innovative parameter-efficient fine-tuning method to increase the adaptability of large language models. IA3 introduces trainable parameters into the attention mechanism of the model so that the model can adapt more precisely to task-specific inputs. It changes the way how input features are processed through attention layers, which enables better capturing and leveraging of useful information by the model. As compared with LoRA which introduces low-rank matrices for adaptation, IA3 can have better adaptability since it focuses on input-aware modification, so these benefits can be particularly helpful for tasks that require a rich context understanding. This high-level adaptation eases the burden and in the meantime does not increase computation complexity too much, hence, IA3 is an ideal option for situations with limited resources or when rapid adaptation is needed.

2.2.3. Prompt tuning:

Prompt tuning involves the process of prepending virtual tokens to the input tokens of the LLM [13]. During fine-tuning, the virtual tokens which are differentiable are learned through back propagation, while the rest of the LLM remains frozen. The approach introduces a series of learnable prompt tokens that are prepended to the input data, allowing the model to leverage its pre-existing knowledge while focusing on task-specific information conveyed through these prompts. By optimizing only the prompt tokens, prompt tuning significantly reduces the computational resources and memory required compared to full model fine-tuning. This method is particularly advantageous for scenarios where rapid adaptation to new tasks is needed, as it enables efficient and effective model customization without extensive retraining of the entire network.

2.2.4. Prefix tuning:

Virtual tokens are inserted by Prefix tuning in all the layers of target model, so it requires more parameters to be learned. Prefix tuning uses auxiliary learnable prefix tokens prepended to the input of a model and trains the model to generate the target sequence starting from those prefix tokens. Since model training happens for a few steps with a small number of parameters, it is much faster than fine-tuning. The advantage of the technique is that none or fewer samples are needed for training – which can be expensive in many applications – while at the same time achieving competitive performance. Notably, better prefix tuning transfer occurs when models are limited in capacity, as finetuning might lead to forgetting previously learned information. Thus, it provides a good option in scenarios where it involves fine-tuning or nothing options and if you want to make decisions very quickly.

2.2.5. QLoRA (Quantized Low-Rank Adaptation):

QLoRA combines LoRA with model quantization, enabling the fine-tuning of LLMs with less GPU memory by reducing the precision of floating point data types within the model [13]. By integrating quantization with low-rank adaptation, the approach compresses the representation of the model's trainable parameters, thus minimizing the storage and processing power required. Despite the quantization, QLoRA ensures that the model's task-specific performance remains robust, offering a scalable and cost-effective approach for rapidly adapting large language models to new applications.

2.3. Mathematical Equation for loss function in fine-tuning Large Language Models (LLMs):

During fine-tuning large language models, a standard autoregressive cross-entropy loss function is minimized as: [13]

$\mathscr{L} = \sum_{i=1}^{T+1} Mi \,.\ \log P(xi \mid x < i),$

where :

$$Mi = \begin{cases} 1\,, & if\ x1 \neq -100 \\ 0, & otherwise. \end{cases}$$

Metrics for evaluation:

The complexity of a code edit task is reflected using the number of differing lines and their edit ratio in the input/output pair. The computation is defined as:

n diff = |(I ∪ O) / (I ∩ O) |

r diff= n diff / |I U O |

[12] where I and O are sets of input/output code with single lines as elements.

The effectiveness of the models is measured through widely used metrics such as the Exact Match (EM)[13] and CodeBLEU metrics [13].

To evaluate the effectiveness of the models on a list of $k \in [1, 10]$ candidates, the **EM@k** is reported, which computes the average correct predictions among a list of k candidates [13].

The hyperparameter value for each PEFT technique[13] have been used in the initial paper are as follows:

For LoRA and IA3, the low-rank matrix decomposition is applied on the attention layers of the models and set r = 16 and α = 32. For implementing QLoRA, 8-bit and 4-bit quantization is used. The learning rate is set to $3e-4$ for LoRA, IA3 and QLoRA. For Prompt tuning and Prefix tuning, learning rates of $3e-3$ and $3e-2$ are applied [13].

3. PROMPT DESIGNING:

3.1. Introduction to Prompt Engineering :

Prompt engineering is the process of designing and refining input prompts to optimize the output of language models, particularly in the context of tasks such as code generation. This involves crafting the prompt in a way that leverages the strengths of the model, addresses potential limitations, and guides the model to produce more accurate, relevant, and contextually appropriate responses.

3.2. Role in Code Generation :

The role of prompts in code generation is crucial, as they directly influence the quality and accuracy of the generated code. Based on the papers, the significance of prompts in code generation can be summarized as follows:

Guidance and Context:

Prompts serve as the primary means of guiding the language model on what specific task to perform. In code generation, a well-crafted prompt can provide clear instructions, specify the context, and outline the expected structure of the output, which helps in generating more accurate and contextually relevant code.[1]

Performance Enhancement:

The structure and wording of the prompt can significantly affect the performance of the model. Different types of prompts, such as task prompts, context prompts, and processing prompts, can be combined to optimize the code generation process.[1] The right combination of prompts has been shown to substantially improve evaluation metrics like BLEU[1] and CodeBLEU[1], which measure the accuracy and quality of the generated code.[1]

Influencing the Model's Output:

Prompts are instrumental in influencing the behavior of the model. By carefully designing the prompt, developers can steer the model toward producing code that is syntactically correct, semantically appropriate, and aligned with the specific requirements of the task at hand.[2]

3.3. Designing effective prompts :

Designing effective prompts for code generation involves a strategic approach to ensure that the language model produces accurate and relevant code. Based on the insights from the research papers, the following is the way how we can design effective prompts:

1. Understand the Task:

Clarity: Be clear about what you want the model to accomplish. If you're working on a Text-to-Code (T2C)[1] task, ensure that the prompt clearly describes the functionality or the outcome you expect from the code.

Context: Provide sufficient context so that the model understands the environment in which the code will operate. For example, specify the programming language, libraries, or specific functions that need to be used.[1]

2. Use Specific Instructions:

Task Prompts: Begin with a task prompt that clearly states the action required. For example, "write a Java method that + #{NL}"[1]

Processing Prompts: For instance, ensure that both positive and negative integers are handled by the function appropriately.

3. Incorporate Contextual Information:

Context Prompts: Include information that provides background or additional context for the task. For example, "Remember you have a Java class named '#{CN}', member variables '#{MV}', and member functions '#{MF}'". [1]

Combining Prompts: Experiment with combining different types of prompts (e.g., task, context, and processing prompts) to see how they interact and influence the model's output. The right combination can significantly improve the quality of the generated code.[1]

4. Iterate and Refine:

Multi-step Optimization[1]: The first step to start with basic prompt and based on output refine it. Ask the model for suggestions on how to improve the prompt and iteratively test different versions.

Testing and Evaluation: Use metrics like BLEU and CodeBLEU to evaluate the output of your prompts. Refine the prompt based on these evaluations to optimize performance.[1]

5. Leverage Existing Code:

Example-based Prompting: Similar code snippets or reference existing code can be used as template to help the model generate code that is more aligned with the expectations.[2]

6. Consider the Model's Limitations:

Simplicity: As it is important to be specific, avoid overly complex prompts that might confuse the model. Keep the instructions clear and concise.[1]

Avoiding Ambiguity: Make sure the prompt does not leave room for multiple interpretations. Ambiguity can lead to unexpected or incorrect code generation.[1]

3.4. Prompt Variation:

Following are various variations or types of prompt :

1. Direct Instruction Prompts[3]:

Direct instruction prompts provide clear and concise requests for specific code functionalities. This straightforward approach often yields good results for simple tasks. For instance, the prompt given for the Snake Game was: "Using the Python language to help me fulfill the requirements of writing a Snake game, please provide a code example." [3]

This prompt is direct and specifies the programming language and the desired functionality, making it easier for the model to generate accurate code.

2. Contextual Prompts:

Contextual prompts provide additional information or context that can help the model understand the requirements better. This approach is particularly useful for more complex tasks.[3]

Example: "Your task is to use the Python language to help me complete the requirements in the triple quotes. Please provide a code example."[3]

By framing the prompt to include the role of a developer, the model can tailor its response more effectively.[3]

3. Multi-step Prompts

Multi-step prompts involve breaking down a task into smaller, manageable parts, allowing the model to generate code iteratively. This method has been shown to improve performance in solving complex problems.[1]

Example: "write a Java method that + #{NL}."

“remember you have a Java class named + ‘#{CN}’, member variables + ‘#{MV}’, member functions + ‘#{MF}’ “.[1]

This approach encourages the model to focus on each component of the task, leading to more structured and coherent code.

4. Specification-based Prompts

Specification-based prompts include detailed requirements or constraints that the code must meet. This method helps ensure that the generated code adheres to specific guidelines.[1]

Example: “Move the snake up, down, left, or right using the arrow keys or WASD keys to make it eat food. Every time the snake eats food, its length increases and it earns points. If the snake hits the game border or itself, it dies and the game ends.”[3] This prompt specifies both the functionality and error handling requirements, guiding the model to produce more robust code.

5. Recursive Criticism and Improvement (RCI):

The RCI technique involves prompting the model to critique its own output and suggest improvements. This iterative refinement can enhance the quality of the generated code. This method encourages self-assessment, leading to potentially more optimized code solutions.[3]

3.5. Prompt Design :

3.5.1. Prompt Design for Text-to-Code Generation :

Prompt description:

In text-to-code (T2C) generation tasks, the objective is to generate code from natural language descriptions.Consider a basic prompt designed to elicit the desired code based on a given description. Initially, the prompt was: "write a Java method that + #{NL}"[1]. Testing this prompt on a sample of 100 instances from training data yielded a performance with a BLEU score of 5.29 and a CodeBLEU score of 22.76[1].

Multi step optimization :

To enhance prompt effectiveness, we employed a multi-step optimization approach:

1. Contextual Enhancement: Feedback from the language model indicated that adding more specific details of the code's expected behavior and the programming context will improve results. Thus, updated the prompt to include information about the code environment, such as context and requirements, with the revised prompt: "remember you have a Java class named: #{CN},member variables + ‘#{MV}’, member functions + ‘#{MF}’"[1]. This modification improved the performance, resulting in a BLEU score of 10.42 and a CodeBLEU score of 25.05[1].

2. Processing Instructions: Further improvements were made by aligning the output with pre-processed ground-truth data. This involved removing comments, method modifiers, and renaming variables according to a specific pattern. A processing prompt was added with instructions such as: "comments should be removed; summary should be removed; throws should be removed; function modifiers should be removed; the method name should be changed to 'function'; argument names should be changed to 'arg0', 'arg1'...; local variable names should be changed to 'loc0', 'loc1'...". This adjustment resulted in better performance, achieving a BLEU score of 13.11 and CodeBLEU score of 36.00[1].

3. Behavior Specification: To address specific API and exception handling requirements, the study utilized prompts to extract these details from the generated code and adjusted the prompt accordingly. We replaced the original task prompt with: "write a Java method #{that calls ...} with[out] exception handling to #{NL}"[1]. Depending on whether APIs and exception handling were required, this prompt was further refined. The final results showed significant accuracy improvements, with BLEU scores of 22.14 and 27.48, and CodeBLEU scores of 44.18 and 46.78, respectively[1].

Iterative refinement of prompts, including the addition of contextual details, processing instructions, and behavior specifications, significantly improved the accuracy of text-to-code generation tasks. This illustrates the effectiveness of a structured and detailed approach to prompt design in enhancing model performance.[1]

3.5.2. Prompt Design for Code-to-Code Generation:

The process for designing prompts for code-to-code (C2C) generation is similar to text-to-code (T2C) generation but with key differences. For C2C generation, consider example to translate a code function from C# to JavaThe prompt initially used: "translate C# code into Java code: #{Code}"[1]. Testing with this prompt resulted in a BLEU score of 9.76 and a CodeBLEU score of 39.37[1].

Multi-Step Optimizations:

Processing Instructions: Unlike T2C, C2C does not involve class-related context or pre-processed ground-truth. The generated code often includes annotations which are not present in the ground-truth. To address this, a processing prompt was added: "do not provide annotation"[1]. This improved CodeBLEU to 45.28 but BLEU only slightly to 8.55[1].

Formatting Adjustments: Updating the prompt to include markdown syntax (e.g., changing #{Code} to "'#{Code}'") further enhanced accuracy, achieving BLEU=15.44 and CodeBLEU=45.00[1].

Behavior Specification: Similar to T2C, requirements for API usage and exception handling were extracted and included in the prompt. The updated prompt was: "translate C# code into Java code: "'#{Code}'" #{that calls ...} with[out] exception handling". This adjustment led to a slight increase in CodeBLEU (46.17) but a reduction in BLEU (8.90) [1].

While adding detailed instructions and requirements can improve certain aspects of code generation, excessive requests may introduce uncertainty, negatively affecting overall performance.[1]

3.6. Challenges and solutions

1. Ambiguity in Natural Language Descriptions

Natural language descriptions often lack precision, leading to potential misunderstandings in code generation. To address this, provide clear and detailed prompts with specific instructions and examples. Using structured formats or templates can help reduce ambiguity and guide the model more effectively.

2. Code Syntax and Style Differences

Different programming languages have distinct syntax and conventions, making it challenging to translate or generate code accurately. Specify the target language's syntax and style requirements in the prompt. Providing examples of properly formatted code can also help guide the model.

3. Handling Complex Requirements

Complex tasks that involve multiple steps, conditional logic, or specific APIs can be difficult to encapsulate in a single prompt. To manage this, break down complex tasks into smaller, manageable components. Use multi-step prompts or iterative refinement to address each part of the task systematically.

Prompt engineering is essential for improving code generation. Well-designed prompts enhance accuracy by providing clear guidance and context. Addressing challenges like ambiguity and syntax differences involves using specific instructions and examples. By iterating on prompts and strategically varying details, developers can significantly boost code quality and relevance.

4. UNDERSTANDING CONTEXT IN LLMS:

Context in large language models (LLMs) refers to the information that the model uses to generate responses. This can include task specifications, retrieved documents, previous conversations, and even model self-reflections, functioning similarly to episodic memory. [6]

4.1. In-Context Learning:

In-context learning (ICL) approach allows models to perform various tasks by conditioning on a few examples provided in the input context, without any parameter updates or fine-tuning.
In-context learning is referred to as the ability of a model to be conditioned on a sequence of input-output pairs (demonstrations) along with a new query input, and the corresponding output is generated. ICL has proven to be helpful in improving the performance of LLMs in many coding related tasks such as code generation[4], and code repair[5].

In the paper titled "Large Language Model-Aware In-Context Learning for Code Generation" (Li et al., 2023)[7], the authors presented a novel approach based on in-context learning called LAIL (Large Language Model-Aware In-Context Learning). LAIL leverages the inherent capabilities of LLMs to estimate the prediction probability of ground-truth programs based on given requirements and example prompts. By evaluating these probabilities, the framework classifies examples into positive and negative categories, thereby providing valuable feedback that informs the selection process. During the inference stage, LAIL operates by taking a specific test requirement and selecting a curated set of examples to serve as prompts for the LLM. [7] This selection is facilitated by a neural retrieval mechanism that has been trained on labeled data, which aims to align closely with the preferences exhibited by LLMs. The preferences of LLMs are understood and learned from, allowing examples to be effectively selected by the retriever from the training set that are most likely to aid in producing correct outputs.

A key innovation of LAIL is its departure from heuristic methods traditionally used for example selection. Instead, it employs a more systematic approach to choose a limited number of high-quality examples that yield superior metric scores when matched against a test requirement.[7] The LAIL approach enhances the performance of In-Context Learning(ICL) by ensuring that the selected examples are highly relevant and conducive to successful code generation.

Another study by Pei et al. (2023)[8], discusses the problem that traditional code completion models often lack comprehensive context, limiting their effectiveness, especially for tasks like function call argument completion.[8] Function call argument completion is defined as the task of predicting the arguments for a given function call based on available context local, project-level, or beyond. The paper introduces a new dataset, CALLARGS, and a development environment, PYENVS, to provide a richer context for code completion models.

The CALLARGS dataset was constructed specifically for function call argument completion, providing comprehensive details about each function call instance. The dataset includes parsed Python files transformed into abstract syntax trees (ASTs), enabling the extraction of crucial elements such as function names, argument locations, and local contexts surrounding function calls. [8]

The paper defines two variations of the function call argument completion task: unidirectional prediction, which simulates writing code from start to finish with only local context available, and in-filling prediction, which Simulates code editing where both preceding and following code is accessible. [8] Additional context of the function implementation information and function usage information is provided for function call argument completion. The functional implementation context reveals the format, constraints, and intention of the current function call. On the other hand, local contexts surrounding the calls of the same function within the project are collected by the function usage context. [8] The usage context offers project-specific examples that help the model better understand and induce the usage for the current function call. A similarity-based ranking criterion was designed to enhance the effectiveness of function call argument completion. It ensures that only the top usages are selected to provide the model.

This research underscores the significance of comprehensive context in improving code language models, particularly in the domain of function call argument completion.

4.2. In-file and Cross-file Context:

In-file context refers to the information and code present within a single file. When a code completion tool uses in-file context, it relies solely on the contents of the current file to predict and suggest code completions. Cross-file context, on the other hand, involves information from multiple files within the same project. This context is crucial in modern software development, where projects are often divided into multiple files and modules, as discussed in the Modular Programming design approach [9].

Traditional code generation models primarily rely on in-file context, neglecting the potential benefits of cross-file information. To address this limitation, recent research by Ding et.al. (2023) has explored the incorporation of cross-file context to enhance code generation accuracy and efficiency.

The researchers developed the tool called Cross-file Context Finder (CCFinder) [10] which is designed to locate and retrieve the most relevant cross-file context. CCFinder works by parsing the project hierarchy and code components to extract project information. It builds a project context graph that represents the details of each component (entity) and the interactions among them (relations). The process of building the project context graph involves creating a root node for the project and connecting it with all file nodes which further builds its own sub-graph. Nodes within the file-level sub-graph link to others based on dependencies or scope. For context retrieval, the closer a graph neighbor is to a specific node, the more relevant that neighbor is.

By incorporating the CCFinder tool, researchers proposed the CoCoMIC framework[10]. The framework uses an autoregressive LM to encode in-file code snippet and retrieved cross-file context, and predicts the next code token conditioning on both. [10] It includes two main parts in its input representation: the source code sample (S) and its cross-file context (C). For entity representation, the framework uses Mean Pooling [SUM] tokens to learn the summarization of each entity.[10] When completing code, the model attends to the representations of these [SUM] tokens for each cross-file entity. The model fuses the in-file and cross-file context at each Transformer layer, ensuring that generating the next token's hidden state always depends on both contexts.

By encoding the code sequence of an entity into a single token, CoCoMIC enables the model to incorporate more cross-file context while saving input length. Finder can retrieve most of the cross-file context that helps the language model complete the input code, increasing identifier recall by 27.07%. [10]

In the paper titled "Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context" (Li et al., 2024) researchers proposed framework, IDE Coder that leverages the native static contexts available in Integrated Development Environments (IDEs) for cross-context construction and utilizes diagnostic results for self-refinement.

One of the primary challenges in dealing with cross-file contexts is ensuring accuracy and relevance. Accuracy involves correctly identifying references and definitions across different files, while relevance pertains to identifying context that accurately reflects the