Explaining Code Generation AI Models: Are We Meeting Stakeholder Needs? Supplementary Material - Prompts

January 17, 2026

1 Prompt 1.0 and Response

Prompt: Imagine you are a software developer and using a code generation AI tool. Apart from providing code, what kind of explanation would you like the tool to provide?

Response: As a software developer using a code generation AI tool, I would appreciate several types of explanations beyond just the generated code itself. These explanations can help me understand how the AI model makes decisions and provide insights into the rationale behind the generated code. Here are some types of explanations I would find valuable:

Explanation of Code Structure: Providing an explanation of the overall structure and organization of the generated code, including the purpose of each component, the flow of control, and the relationships between different modules or functions, helps me understand the high-level design decisions made by the AI model.

Explanation of Code Patterns: Explaining the underlying code patterns or design principles used by the AI model to generate code snippets can help me understand the reasoning behind the code generation process. This could include insights into common programming idioms, best practices, and architectural patterns reflected in the generated code.

Contextual Information: Providing contextual information about the input data, such as the input specifications, requirements, constraints, or user intentions, helps me understand the context in which the code was generated and interpret its relevance and suitability for the given task or problem.

Explanation of Variable Names and Comments: Offering explanations for variable names, function names, and comments included in the generated code can provide insights into the semantics and intentions behind the code constructs. Clear and meaningful naming conventions and descriptive comments enhance code readability and facilitate comprehension.

Explanation of Code Dependencies: Explaining the dependencies and external libraries used by the generated code, as well as their purpose and relevance to the task at hand, helps me understand the external resources required for the code to function properly and assess any potential dependencies or integration issues.

Alternative Code Options: Presenting alternative code options or variations generated by the AI model, along with explanations for why each option was considered and the trade-offs involved, allows me to explore different implementation choices and make informed decisions based on the provided insights.

Model Confidence and Uncertainty: Providing estimates of model confidence or uncertainty associated with the generated code, as well as any limitations or caveats associated with the predictions, helps me assess the reliability and trustworthiness of the generated code and make informed judgments about its suitability for deployment.

By incorporating these types of explanations into the code generation AI tool, I can gain deeper insights into the generated code, understand the reasoning behind the AI model's decisions, and make more informed decisions about code adoption, customization, and integration into my software projects. Additionally, transparent and interpretable code generation models foster collaboration among developers, facilitate knowledge sharing, and promote a culture of continuous improvement in software development practices.

2 Prompt 2.0 and Response

Prompt: Now imagine you are a model engineer and creating an AI based code generation model. What kind of model interpretation would you like and how can it benefit you?

Response: As a model engineer developing an AI-based code generation model, having robust model interpretation capabilities is crucial for understanding the model's behavior, diagnosing performance issues, and improving its effectiveness. Here are some types of model interpretation that I would find valuable and how they can benefit me as a model engineer:

Attention Mechanisms: Understanding the attention mechanisms or similar mechanisms used by the model to focus on relevant parts of the input data during code generation can provide insights into how the model makes decisions. Visualizing attention weights or attention maps can help me understand which parts of the input data are most influential in generating specific code tokens.

Code Structure Visualization: Visualizing the internal representation of the code structure learned by the model, such as abstract syntax trees (ASTs) or code embeddings, can provide insights into how the model captures the hierarchical structure and semantic relationships of code snippets. This visualization can aid in debugging model architecture issues and understanding the learned representations.

Generation Log: Logging the generation process, including intermediate steps, decisions, and probabilities associated with each generated token, allows me to trace the model's behavior and diagnose performance issues. Analyzing the generation log can help identify patterns, biases, or errors in the model's output and guide model refinement efforts.

Feature Importance Analysis: Identifying which input features or representations are most influential in generating specific code tokens or structures can help me understand the factors driving code generation decisions. Feature importance analysis can guide feature engineering, input preprocessing, and model optimization strategies to enhance the model's performance.

Error Analysis: Analyzing errors made by the model, such as incorrect predictions, syntactic errors, or semantic inconsistencies in the generated code, helps me diagnose model weaknesses and identify areas for improvement. Understanding the types and patterns of errors made by the model can inform targeted error correction techniques and model updates.

Domain-Specific Insights: Incorporating domain-specific knowledge or rules into the model interpretation process can provide insights into how the model performs in specific programming languages, software engineering contexts, or application domains. This domain-aware interpretation can inform model refinement and customization for different use cases and scenarios.

Model Debugging Tools: Developing tools and utilities for debugging the code generation model, such as techniques for analyzing model gradients, inspecting internal model states, and visualizing model predictions, can streamline the model development process and accelerate troubleshooting efforts. By incorporating these model interpretation capabilities into the code generation model development process, I can gain a deeper understanding of the model's behavior, diagnose performance issues, and iteratively improve the model to better meet the needs of software developers and practitioners. Additionally, transparent and interpretable code generation models foster collaboration among team members, facilitate knowledge sharing, and promote a culture of continuous improvement in software development practices.