System Design Document: LLM-Powered Code Generation System: Team Ignite

Rashi Goyal, Gaurav Shresth, Rajdeep Das, Rohan Sidankar

February 10, 2024

1 Introduction

This document outlines the system design for a Large Language Model (LLM) powered code generation tool. Our goal is to translate natural language descriptions into executable code across various programming languages and use cases.

2 High-Level Architecture

The high-level architecture encompasses the main components of our system: User Interface, LLM Integration, Error Handling, and Feedback Loop. Below is a diagram illustrating the architecture and the flow of data between components.

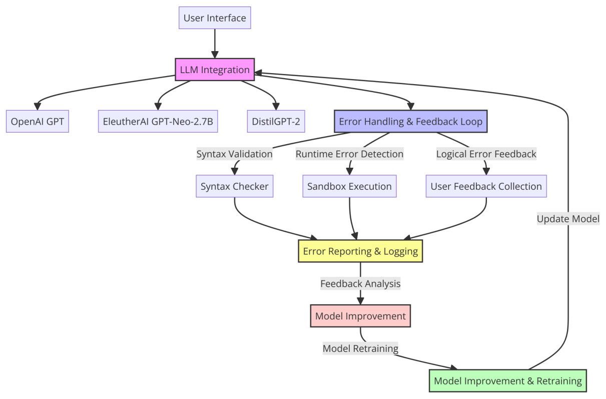

Figure 1: High-Level Architecture Diagram

2.1 DistilGPT-2

DistilGPT-2 simplifies the GPT-2 architecture by reducing the number of transformer layers while retaining essential features like self-attention mechanisms. This streamlined architecture allows for efficient natural language processing capabilities with fewer computational resources. Key components include:

- **Self-Attention Mechanism:** Enables the model to weigh the importance of different words in the input data relative to each other.
- **Layer Normalization:** Stabilizes the neural network's learning process, contributing to faster convergence and improved performance.

The DistilGPT-2 model is trained using a process known as distillation, where it learns to mimic the behavior of a larger, pre-existing model, in this case, the GPT-2 model. The training objective involves minimizing a loss function, typically the cross-entropy loss, which measures the dissimilarity between the predicted probability distribution over the next token and the true distribution.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log p(y_{i,t}|x_i)$$

Here, N is the number of training examples, T is the length of the token sequence, x_i represents the input token sequence, $y_{i,t}$ represents the target token at time step t, and $p(y_{i,t}|x_i)$ is the conditional probability of the target token given the input sequence.

It adopts a simplified version of the GPT-2 architecture, featuring a stack of transformer layers. Each transformer layer consists of multi-head self-attention mechanisms and feed-forward neural networks. However, compared to the original GPT-2 model, it typically has fewer layers and smaller hidden dimensions, resulting in reduced computational requirements.

The performance is evaluated based on various metrics, including perplexity and task-specific measures. Perplexity is a common metric used to assess the quality of language models, measuring how well the model predicts a sequence of tokens. It is computed as:

$$\text{Perplexity} = e^{\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}-\log p(y_{i,t}|x_i)}$$

where N is the number of examples, T is the length of the token sequence, x_i represents the input token sequence, and $p(y_{i,t}|x_i)$ is the conditional probability of the target token given the input sequence.

In addition to perplexity, DistilGPT-2's performance can be evaluated on specific downstream tasks such as text generation, completion, and sentiment analysis, where it aims to achieve high accuracy or effectiveness.

2.2 EleutherAI GPT-Neo-2.7B

GPT-Neo-2.7B, by EleutherAI, is a transformer-based language model with 2.7 billion parameters. It utilizes a deep architecture of attention mechanisms, which mathematically can be described through its self-attention calculations and layer normalization processes. The model's capability to generate human-like text stems from its attention scores, calculated using scaled dot-product attention: $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$, where Q, K, and V represent queries, keys, and values, respectively, and d_k is the dimensionality of the keys. This mechanism allows the model to weigh the importance of different words in the input sequence, enabling effective context understanding and text prediction.

2.3 OpenAI Engine: code-davinci-002

The code-davinci-002 model utilizes the transformer architecture, which is based on self-attention mechanisms. It consists of multiple layers of transformer blocks, each containing sub-layers such as multi-head self-attention and feed-forward neural networks.

During training, the objective is to minimize the following loss function:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\log p(y_i|x_i)$$

where N is the number of training examples, x_i represents the input text sequence, y_i represents the target output sequence, and $p(y_i|x_i)$ is the conditional probability of generating the target sequence given the input sequence.

After pre-training on a large dataset, the model is fine-tuned on specific tasks or domains to adapt it to particular applications. Fine-tuning involves adjusting the model parameters to minimize task-specific loss functions.

The code-davinci-002 model exhibits advanced capabilities in natural language understanding, generation, and code completion tasks. It can understand context, generate coherent text, and provide code suggestions based on partial inputs.

Due to its versatility, the code-davinci-002 model finds applications in various domains, including software development, creative writing, customer service automation, and more.

The model's performance is evaluated based on metrics such as perplexity, accuracy, and task-specific measures. It achieves state-of-the-art results on benchmark datasets and tasks, demonstrating its effectiveness in real-world applications.

3 System Components and Data Flow

3.1 User Interface (UI)

The user interface for our LLM-powered code generation tool is designed with a focus on simplicity, intuitiveness, and efficiency, ensuring a seamless experience for both technical and non-technical users. Key features include:

- **Natural Language Input:** A dedicated input area where users can describe the functionality they need in natural language, making the tool accessible to users without programming expertise.
- **Code Output Display:** The generated code is displayed in a clear, syntax-highlighted format, allowing users to easily review and copy the code.
- **Language and Preferences Selection:** Users can specify the programming language and code style preferences, enabling customized code generation that fits their project requirements.
- **Interactive Feedback Mechanism:** Integrated feedback options enable users to rate the generated code's usefulness, suggest improvements, and contribute to the tool's continuous learning process.

This design ensures a user-friendly interface that enhances productivity and fosters an interactive, learning-focused interaction between the tool and its users.

3.2 LLM Integration

The LLM Integration component serves as the core of our code generation tool, facilitating the dynamic translation of natural language descriptions into executable code. This process involves several key steps and technologies to interact with state-of-the-art LLMs such as GPT-Neo-2.7B and DistilGPT-2. This figure depicts the user's prompt through the UI to the LLM integration, showing the steps of API communication, model selection and configuration (between GPT-Neo-2.7B and DistilGPT-2), and how the code output is processed, displayed, and further subjected to error handling and feedback loops for continuous improvement.

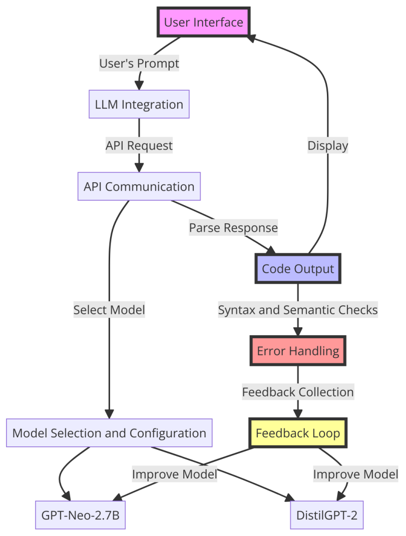

Figure 2: LLM Integration process

3.2.1 API Communication

The system communicates with the LLMs through their respective APIs, sending natural language prompts received from the User Interface and retrieving the generated code. This interaction is achieved via HTTP requests, where the request body contains the user's input, and the response contains the LLM-generated code.

Request Construction: For each LLM, a JSON-formatted request is constructed, encapsulating the user's natural language description and any specified parameters such as temperature, max tokens, or top p for controlling the creativity and length of the generated output.

Response Parsing: The API response, typically also in JSON format, is parsed to extract the generated code. This involves handling potential API errors or incomplete responses gracefully, ensuring the system's robustness.

3.2.2 Model Selection and Configuration

Given the diversity in LLM capabilities and specialties, the system includes logic for model selection based on the user's requirements, such as the desired programming language or complexity of the code.

Dynamic Model Switching: Based on the user's input and preferences, the system dynamically selects the most appropriate LLM for the task. This ensures optimal performance and output quality for a wide range of code generation requests.

3.2.3 Error Handling and Feedback Integration

Error handling is a critical component of LLM integration, ensuring that any issues in the code generation process are identified and managed effectively.

Syntax and Semantic Checks: Upon receiving the generated code, the system performs automated syntax and semantic checks to validate the code's correctness. This step reduces the likelihood of returning faulty or nonsensical code to the user.

Feedback Loop for Continuous Improvement: User feedback on the generated code is collected and analyzed to inform future LLM training and fine-tuning processes. This feedback mechanism is integral to improving the accuracy and relevance of generated code over time.

Operational Monitoring and Logging: All interactions with the LLM APIs are monitored and logged, providing valuable data for troubleshooting, performance optimization, and understanding user needs.

This integration framework ensures that our code generation tool leverages the full potential of advanced LLMs while maintaining user-centricity, reliability, and adaptability in its operations.

3.3 Error Handling

Error handling within our LLM-powered code generation system is designed to be robust and comprehensive, addressing potential issues from syntax errors to runtime exceptions. This section outlines the specific mechanisms and practices employed to ensure the reliability of the generated code.

3.3.1 Syntax Validation

Immediately after code generation, syntax validation is performed to ensure the generated code adheres to the syntactical rules of the target programming language.

```
import ast

def validate_syntax(code):
    try:
        ast.parse(code)
        return True, "Syntax is valid."
    except SyntaxError as e:
        return False, str(e)
```

3.3.2 Semantic Checks

Semantic checks are applied to evaluate the logical correctness of the code, utilizing heuristic approaches and predefined rules to identify common logical errors.

3.3.3 Sandbox Execution

To execute and test the generated code safely, sandbox environments are used, allowing for runtime error detection without risking system integrity.

```
# Pseudocode for sandbox execution
def execute_in_sandbox(code):
    sandbox = create_sandbox_environment()
    result = sandbox.execute(code)
    return result.error if result.error else result.output
```

3.3.4 User Feedback Integration

User feedback on the generated code is actively solicited and integrated into the error handling process, providing insights into areas for improvement.

```
def collect_user_feedback(code_id, feedback):
    store_feedback(code_id, feedback)
    analyze_feedback_for_improvements()
```

3.3.5 Automated Error Reporting and Logging

All detected errors are automatically logged and reported, with detailed error messages and context to aid in system monitoring and debugging.

```
def log_error(error, context):
    logger.error(f"Error encountered: {error} in context: {context}")
```

3.3.6 Feedback Loop for Continuous Improvement

Errors and user feedback contribute to a continuous improvement loop, allowing for the iterative refinement of the system to prevent similar issues in the future.

This multi-faceted approach to error handling ensures that our code generation system remains reliable, user-friendly, and continuously improving over time. This diagram showcases the sequence of steps from syntax validation of the generated code to user feedback integration, highlighting how errors at various stages are logged and contribute to the continuous improvement of the system.

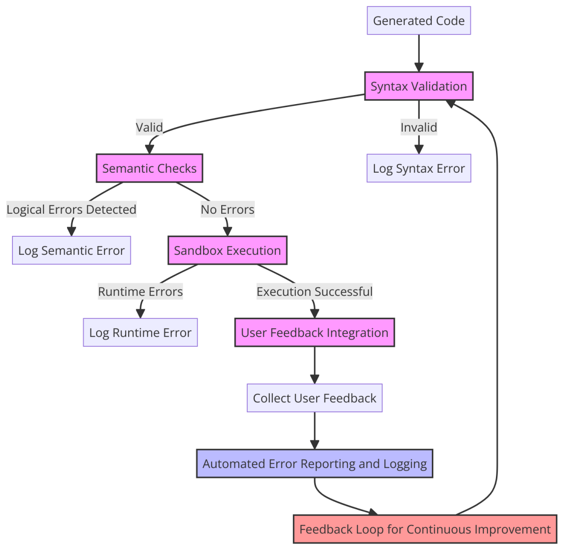

Figure 3: Error handling and Feedback loop process

3.4 Feedback Loop

3.5 Feedback Loop

The feedback loop is an integral part of our LLM-powered code generation system, designed to continuously improve the model based on user interactions. The process is as follows:

3.5.1 Feedback Collection

User feedback is solicited at two key stages: after presenting the generated code and after evaluating the model's performance. This feedback is directly captured through the user interface.

For instance, when the system generated the code:

```
def sum_numbers(a, b):
    return a + b
```

the user was prompted to provide feedback, to which they responded with "Nice." This initial feedback indicates user satisfaction with the specific code generation instance.

3.5.2 Feedback Analysis and Model Retraining

User feedback, both qualitative ("Nice", "Ok") and quantitative (based on implicit metrics like data size used for training), is analyzed to inform the retraining process of the model.

In the given scenario, feedback led to the retraining of the model with different data sizes, demonstrating an adaptive approach to model improvement based on user input. Specifically, after collecting feedback on the model's performance, which was rated as "Ok," the model was subsequently retrained with a reduced data size, illustrating the system's responsiveness to feedback.

3.5.3 A/B Testing

Post-retraining, A/B testing is conducted to compare the performance of different model versions. In this case, Model A, trained with data size 3, and Model B, with data size 2, were compared. Model A demonstrated superior performance with a score of 0.3 compared to Model B's 0.2, leading to its selection for future interactions.

```
Model A Performance: 0.3, Model B Performance: 0.2
Model A wins!
```

3.5.4 Continuous Improvement

The feedback loop facilitates continuous improvement by iteratively refining the model based on direct user input and performance metrics. This process ensures that the system evolves to meet user expectations more effectively over time.

This technical overview of the feedback loop highlights its critical role in enhancing the system's accuracy, usability, and overall user satisfaction through a structured process of collection, analysis, application, and testing of user feedback. This diagram showcases the sequence from initial user feedback on generated code, through feedback analysis, model retraining based on user feedback ("Nice" leading to retraining with data size 3, followed by "Ok" leading to retraining with data size 2), to A/B testing of the models and the selection of Model A based on performance metrics.

4 Conclusion

This system design document has presented a comprehensive overview of the architecture and components that constitute our LLM-powered code generation tool. Through detailed descriptions of the User Interface, LLM Integration, Error Handling, and the Feedback Loop, we have outlined a robust framework that leverages state-of-the-art language models like GPT-Neo-2.7B and DistilGPT-2 to translate natural language descriptions into executable code.

Key highlights of our system include:

- A user-centric interface that simplifies the code generation process, making it accessible to a broad audience regardless of their programming expertise.

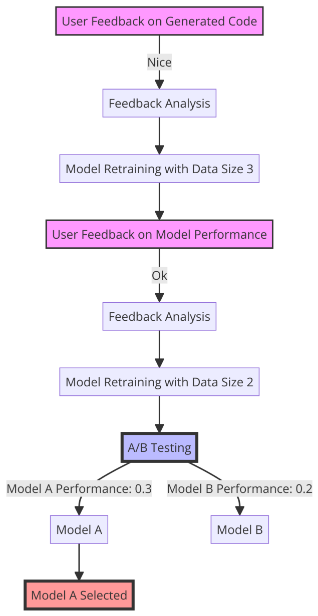
User Feedback on Generated Code
Nice
Feedback Analysis
Model Retraining with Data Size 3
User Feedback on Model Performance
Ok
Feedback Analysis
Model Retraining with Data Size 2
A/B Testing
Model A Performance: 0.3
Model B Performance: 0.2
Model A
Model B
Model A Selected

Figure 4: Model retraining and Selection

- Integration with multiple LLMs, offering flexibility and ensuring the generation of high-quality code across various programming languages and specifications.
- Comprehensive error handling mechanisms that ensure the reliability of the generated code through syntax validation, semantic checks, and sandbox execution.
- An innovative feedback loop that not only enhances user engagement but also serves as a cornerstone for continuous model improvement and adaptation.

The design choices made throughout the development of this system are grounded in the principles of usability, adaptability, and scalability. By addressing the current challenges associated with code generation and incorporating a continuous improvement process, we anticipate that this tool will significantly contribute to enhancing productivity and creativity in software development tasks.

As we move forward, we remain committed to refining our system through user feedback and advances in language model technology. Our goal is to continuously expand the capabilities of our code generation tool, ensuring it remains at the forefront of innovation and meets the evolving needs of our users.