ZOHAIB AZAM

Rust code generation through llm

Contents

[Experimental Setup 2](#_Toc184738857)

[1. Purpose 2](#_Toc184738858)

[2. Tools and Libraries 2](#_Toc184738859)

[3. Process Flow 3](#_Toc184738860)

[1. Input Handling: 3](#_Toc184738861)

[2. Code Generation: 3](#_Toc184738862)

[3. Compilation and Testing: 3](#_Toc184738863)

[4. Result Logging: 3](#_Toc184738864)

[5. Output: 4](#_Toc184738865)

[4. Experimental Environment 4](#_Toc184738866)

[5. Data Pipeline 4](#_Toc184738867)

[Performance Summary 4](#_Toc184738868)

[Observations 5](#_Toc184738869)

[Insights on Failures 5](#_Toc184738870)

[Future Insights and Solutions for Correct Code Generation 6](#_Toc184738871)

[1. Using GPT's Thread API for Incremental Learning 7](#_Toc184738872)

[2. Real-Time Compiler Feedback for Refinement 7](#_Toc184738873)

[Long-Term Benefits 8](#_Toc184738874)

# Experimental Setup

[Repository](https://github.com/Zohaib58/Benchmarking-LLM4Codes)

## 1. Purpose

The purpose of this experimental setup is to automate the generation, execution, and validation of Rust code based on prompts using OpenAI's language models. The experimental process evaluates the correctness of generated Rust code by executing test cases in a controlled environment.

## 2. Tools and Libraries

* **Programming Language**: TypeScript (via NestJS framework).
* **Human Eval Rust File** – prepared manually inspired from Human Eval OpenAI Dataset

[[](https://khiibaedu-my.sharepoint.com/:x:/g/personal/z_azam_22732_khi_iba_edu_pk/EQVKad3ZqIZIjq4MbaXzR0AB1Gl5H1s6MKDvGiJwgHSlAw?e=hB9CH9)Human Eval Rust.xlsx](https://khiibaedu-my.sharepoint.com/:x:/g/personal/z_azam_22732_khi_iba_edu_pk/EQVKad3ZqIZIjq4MbaXzR0AB1Gl5H1s6MKDvGiJwgHSlAw?e=hB9CH9)

* **Key Libraries and Modules**:
  + xlsx: For reading and writing Excel files to handle test cases and results.
  + fs: For file system operations like creating temporary files and directories.
  + path: For managing file paths.
  + child\_process: For running Rust code inside a Docker container.
  + OpenAI: For interacting with the OpenAI API to generate Rust code snippets.
  + os: For accessing system-level temporary directory utilities.

## 3. Process Flow

The experimental setup follows these steps:

### Input Handling:

* **File Upload**: The system reads an uploaded Excel file containing test cases and prompts. For our experiments, the file is named Human Eval Rust.xlsx.
* **Sheet Parsing**: The first sheet of the uploaded Excel file is parsed and converted into a structured JSON representation. This JSON format serves as the intermediary data format, allowing efficient processing and integration with the evaluation pipeline.
* **Additional Input Parameters**:
  + **Model Selection**: The model parameter (e.g., gpt-3.5-turbo) specifies the version of GPT used for processing the test cases.
  + **Output Configuration**:
    - The newFileName parameter determines the name of the output file for storing the results.
    - The newSheetName parameter specifies the name of the sheet within the output file where processed results will be written.

### Code Generation:

* 1. For each prompt, the code is sent to the OpenAI GPT model specified (gpt-4, etc.).
  2. GPT responds with Rust code snippets based on the input prompt.

### Compilation and Testing:

* 1. The generated code is combined with test cases from the Excel file.
  2. Code is executed within a **Rust Docker container** to ensure a consistent environment.
  3. Output is parsed to determine whether the tests passed or failed.

### Result Logging:

* 1. Results are recorded in the Excel sheet:
     1. Combined code and test case.
     2. Output of the Rust compiler.
     3. Boolean indicator (1 for success, 0 for failure).
  2. Total successes and the percentage of correct outputs are calculated and added to the Excel sheet.

### Output:

* 1. The updated Excel file is saved with additional information.

## 4. Experimental Environment

* **System Setup**:
  + Docker is required to execute Rust code securely and consistently.
  + Rust dependencies (rand, chrono, regex, num, md5) are managed via Cargo.
* **OpenAI API**:
  + The system interacts with OpenAI models via their API using the chat.completions endpoint.

## 5. Data Pipeline

* **Input Data**: Prompts and test cases from an Excel sheet.
* **Processing**:
  + Prompts are sent to OpenAI's GPT model to generate Rust code.
  + Generated code is combined with test cases and executed in Docker.
* **Output Data**: Results are saved back to the Excel file with success indicators.

The experiment was conducted using two different OpenAI models, **GPT-4** and **GPT-3.5 Turbo**, to evaluate their performance in generating and executing Rust code. The results are summarized below:

## Performance Summary

1. **GPT-4**:
   * **Total Prompts**: 164
   * **Correct Outputs**: 145
   * **Accuracy**: **88.41%**
2. **GPT-3.5 Turbo**:
   * **Total Prompts**: 164
   * **Correct Outputs**: 113
   * **Accuracy**: **68.90%**

## Observations

* ***Multiple iterations were carried out to benchmark 164 test cases for both the models. Due to the unpredictable nature of the model , Rust compiler setup hosted on Docker and the testing environment had to undergo multiple changes specially to declare dependencies in Cargo.***
* GPT-4 demonstrated significantly higher accuracy than GPT-3.5 Turbo, suggesting improved capabilities in understanding prompts and generating correct Rust implementations.

# Insights on Failures

**1. Assertion Failures in Test Cases**

These errors indicate mismatched expected and actual outputs in the test cases, often due to logical issues in the generated code:

* **Example:**
  + thread 'tests::test\_longest' panicked
    - **Cause:** Incorrect logic in the implementation resulting in a wrong output comparison (left: Some("z"), right: Some("x")).
  + thread 'tests::test\_sort\_third' panicked
    - **Cause:** Sorting logic did not produce the expected result (left: [9, 3, -5, ...], right: [5, 3, -12, ...]).
  + thread 'tests::test\_generate\_integers' panicked
    - **Cause:** Incomplete range in the generated integers (left: [2, 4, 6, 8, 10], right: [2, 4, 6, 8]).

**2. Arithmetic Errors**

These failures involve operations that are invalid or result in runtime issues:

* **Example:**
  + thread 'tests::test\_get\_max\_triples' panicked
    - **Cause:** Attempt to subtract with overflow, likely due to handling an unsigned integer without sufficient checks.

**3. Mismatched Data Types**

These errors occur when the generated code uses types that do not align with the expected input/output:

* **Example:**
  + error[E0308]: mismatched types
    - **Cause:** Passing a float where an integer was expected in function parameters or return values.

**4. Method or Function Usage Errors**

Generated code attempted to use methods or functions that do not exist or were incorrectly referenced:

* **Example:**
  + error[E0599]: no method named 'rsplit\_whitespace'
    - **Cause:** Incorrect method invocation; split\_whitespace was intended instead of rsplit\_whitespace.
* **Common Errors**:
  + Compilation failures due to missing imports or syntax issues.
  + Logical errors where the generated Rust code doesn't fulfill the test case requirements.
  + Incorrect handling of edge cases (e.g., invalid indices or unexpected inputs).

**Possibilities for Improvement**

1. **Prompt Engineering**:
   * Refine prompts to include specific constraints, edge cases, and expected imports.
2. **Post-Processing**:
   * Automatically validate and fix common Rust errors before compilation.
3. **Extended Validation**:
   * Include additional layers of logical validation for test cases beyond just ok or FAILED.
4. **Dynamic Docker Management**:
   * Optimize Docker container handling to reduce overhead in test execution.

# Future Insights and Solutions for Correct Code Generation

**Please note that the following insights draw inspiration from my past projects and Type Guided Migration Research Paper.**

To achieve more accurate and reliable code generation, leveraging advanced features like GPT's **Thread API** and integrating automated refinement loops with real-time feedback mechanisms from the development environment can be transformative. Here are two specific approaches to address the challenges:

## 1. Using GPT's Thread API for Incremental Learning

The **Thread API** feature allows GPT to maintain context across multiple interactions, effectively "training" a thread for a specific purpose. This capability can be harnessed to refine code generation iteratively, minimizing recurring errors and adapting the model to a particular project's requirements.

**Key Implementation Steps:**

* **Error Learning**: Use the Thread API to inform GPT about specific mistakes to avoid, such as incorrect type annotations in Rust, mismanagement of lifetimes, or unsafe practices.
  + Example: "In this thread, always ensure that Vec initializations in Rust include explicit type annotations, like Vec::<T>::new(), to avoid ambiguity."
* **Incremental Feedback**: Provide the model with corrections for previous outputs, enabling it to improve subsequent generations. Over time, the thread can specialize in generating highly accurate, project-specific code.
  + Example: After identifying and fixing a lifetime issue in Rust, inform the model: "In future responses, ensure that lifetime parameters are explicitly defined where references are used."

**Advantages:**

* Enables a customized "training" loop without modifying the underlying model.
* Reduces recurring errors by adapting to user feedback over the course of a thread.
* Helps refine code quality for specific programming languages, frameworks, or project domains.

## 2. Real-Time Compiler Feedback for Refinement

In an integrated development environment where a **Rust container** is embedded, failed test cases or compilation errors can be utilized as immediate feedback to refine the generated code. This creates a feedback loop between the development environment and GPT, ensuring continuous improvement.

**Key Implementation Steps:**

* **Rust Compiler Integration**: For test cases that fail, capture the **Rust compiler's output** (e.g., borrow checker errors, type mismatches, or syntax issues) and send it back to GPT for analysis and refinement.
  + Example: If the compiler reports error[E0382]: borrow of moved value, the error can be passed to GPT along with the original code snippet. GPT can then adjust the code to fix the issue, such as by cloning the value or adjusting ownership semantics.
* **Test Case Execution**: Run the generated code within the Rust container, automatically executing pre-defined test cases. Any failures (e.g., mismatched outputs or panics) are used to iteratively refine the code.
* **Error Context Feedback**: Pair the compiler error messages with human-readable context to enhance GPT's ability to generate correct fixes.

**Example Workflow:**

1. **Initial Code Generation**: GPT generates Rust code for a given prompt.
2. **Test Case Execution**: The Rust container runs the code against test cases.
3. **Error Feedback Loop**:
   * If a test fails, capture the exact error message (e.g., "expected i32, found String").
   * Provide this feedback to GPT in the thread: "Fix the following type mismatch error in the code: expected i32, found String."
4. **Refined Code**: GPT generates a revised version of the code, addressing the error.

**Advantages:**

* Ensures alignment with the compiler's requirements and test case expectations.
* Reduces the need for manual debugging by automating error detection and correction.
* Improves model understanding of complex language-specific behaviors, such as Rust's borrow checker and ownership model.

## Long-Term Benefits

By combining the **Thread API** for incremental training with **real-time feedback from a Rust-integrated environment**, these solutions offer the potential for:

* **Project-Specific Expertise**: Threads trained for specific projects or domains can significantly enhance GPT's ability to generate relevant and correct code.
* **Automated Error Correction**: A seamless feedback loop between the compiler and GPT minimizes manual intervention, streamlining the development process.
* **Improved Model Understanding**: Repeated exposure to specific error patterns, combined with compiler feedback, can enhance GPT's underlying knowledge and adaptability.

These approaches provide a robust framework for leveraging AI-assisted code generation in real-world software development environments, enabling faster iterations, fewer errors, and higher-quality outputs.

Top of Form

Bottom of Form