Paper ET01

AUTOMATING SAS AND R CODE INTERPRETATION AND DEBUGGING: A PRACTICAL PIPELINE FOR STATISTICAL PROGRAMMERS

Jaime Yan, Merck & Co., Inc., Rahway, NJ, USA

Tingting Tian, Merck & Co., Inc., Rahway, NJ, USA

ABSTRACT

This paper introduces a practical pipeline for automating the interpretation and debugging of SAS and R code, specifically designed for statistical programmers in the pharmaceutical and healthcare industries. The system leverages Large Language Models (LLMs), such as Claude or GPT-4, to translate user queries into executable code, run the code within a Jupyter kernel environment, and automatically debug any resulting errors. The pipeline includes a robust, iterative error-handling mechanism that utilizes the LLM to refine and correct the code until accurate results are achieved. A Gradio-based user interface provides a clear and interactive display of both the generated code and its execution output. This approach significantly improves productivity and accessibility in data analysis by automating code generation, execution, and debugging tasks. We describe the system architecture, workflow, and technical innovations, present a rigorous evaluation of the system's performance, and discuss its limitations and potential for future development. We also provide access to an open-source GitHub repository for implementation and further development.

***K*eywords** Large Language Models (LLMs), SAS, R, Code Automation, Debugging, Statistical Programming, Jupyter Kernel, Pharmaceutical Industry, Data Analysis

1 Introduction

Statistical programming in languages such as SAS and R is integral to data analysis within the pharmaceutical and healthcare industries. Tasks such as data manipulation, statistical modeling, report generation, and visualization frequently involve writing and debugging complex code—processes that are both time-intensive and error-prone [1]. The increasing volume and complexity of data in these fields further exacerbate these challenges, highlighting the need for more efficient and reliable programming tools. Large Language Models (LLMs) offer a promising solution by automating key aspects of the statistical programming workflow.

This paper introduces an automated pipeline that integrates LLMs with SAS and R environments to enhance the efficiency and accuracy of statistical programming. The proposed system automates code generation, execution, and, crucially, iterative debugging, addressing common inefficiencies and error rates inherent in traditional workflows. By leveraging LLMs, such as Claude and GPT-4, our system translates user queries into executable code, runs the code within a Jupyter kernel environment, and systematically refines it through automated debugging. This iterative process not only accelerates data analysis but also improves accessibility for users with varying levels of programming expertise. Additionally, a Gradio-based interactive interface enhances usability by providing a clear and intuitive platform for real-time code execution and debugging.

The remainder of this paper is structured as follows: Section 2 reviews related work on LLMs in statistical programming, automated debugging, and intelligent code generation. Section 3 details the methodology behind our system, including its architecture, workflow, and key technical innovations. Section 4 presents the experimental setup, evaluation metrics, and performance results. Section 5 discusses the implications of our findings, system limitations, and potential future directions. Finally, Section 6 concludes the paper.

2 Related Work

This section reviews the existing literature relevant to our research, covering LLMs in code generation and debugging, automated debugging techniques, statistical programming tools, and applications of LLMs in healthcare and pharmaceutical research.

2.1 LLMs in Code Generation and Debugging

Recent advancements in LLMs have demonstrated their potential in various code-related tasks. Models like Codex [2], AlphaCode [3], and InCoder [4] have shown impressive code generation capabilities from natural language descriptions. However, these models often focus on general-purpose programming languages (e.g., Python, Java) and are not optimized for the specialized syntax and libraries used in statistical programming with SAS and R. While some studies have explored using LLMs for program repair [5, 6], the focus has frequently been on simple syntax errors, rather than the logical and statistical errors that are common in data analysis code. Our work builds upon this research by *specifically* targeting SAS and R and by developing a robust, *iterative* error-handling mechanism tailored to the unique challenges of statistical programming.

2.2 Automated Debugging Techniques

Traditional automated debugging techniques include static analysis (e.g., linting, type checking), dynamic analysis (e.g., runtime error detection, tracing), fault localization (e.g., spectrum-based fault localization, program slicing), and program repair (e.g., genetic programming, rule-based repair) [7, 8, 9]. Techniques like program slicing [10] and symbolic execution [11] have been used to isolate faulty code segments. While these techniques have proven effective in various contexts, they often struggle with the complex logic and domain-specific knowledge required for statistical programming. Critically, many existing tools don't effectively address *logical* errors in statistical code, where the code may be syntactically correct but produce incorrect results due to flawed statistical reasoning. Our approach leverages the reasoning abilities of LLMs to overcome these limitations and address both syntactic and logical errors.

2.3 Statistical Programming Tools

Existing tools for statistical programming in SAS and R, such as SAS Studio, RStudio, and Jupyter Notebooks, provide features like syntax highlighting, code completion, and integrated debugging environments. These tools improve programmer productivity but primarily rely on *manual* debugging efforts. While some offer limited automated assistance, such as suggesting syntax error corrections, they lack the ability to understand the user's *intent* and provide higher-level debugging guidance for both syntactic and semantic errors. Our system complements these tools by automating the more tedious and error-prone aspects of code generation and debugging, allowing users to focus on the statistical aspects of their work.

2.4 LLMs in Healthcare/Pharmaceutical Research

LLMs are increasingly being applied in healthcare and pharmaceutical research, including areas like drug discovery [12], clinical trial design optimization [13], medical text analysis and summarization [14], and personalized medicine [15]. However, the application of LLMs to automate statistical programming tasks within these domains remains relatively unexplored. Our work directly addresses this gap by providing a practical tool that can enhance the efficiency and accuracy of data analysis in pharmaceutical and healthcare research, leading to faster insights and potentially improved patient outcomes.

Table 1 provides a comparative overview, highlighting the unique contributions of our system.

Table 1: Comparison with Existing Work

Approach	Focus	Debugging	Languages	Key Contribution
Codex [2]	Code generation	Limited syntax error handling	General-purpose	General code generation
AlphaCode [3]	Code generation	Minimal	General-purpose	Competitive programming
Traditional Debuggers	Syntax/Runtime Errors	Automated (limited)	Various	Fault localization, breakpoints
RStudio/SAS Studio	IDE	Manual	R/SAS	Syntax highlighting, code completion
Our System	Code generation	**Iterative, LLM-powered**	**SAS, R**	**Automated statistical code debugging**

3 Methodology

3.1 Theoretical Foundation

The integration of Large Language Models (LLMs) with multi-language code interpretation presents unique challenges in system design and implementation. We formalize our approach through a rigorous theoretical framework.

3.1.1 System Model

We define our system as a composite structure $S = (L, K, T, O)$, where:

$$L : \text{LLM interface component} \tag{1}$$
$$K : \text{Set of language kernels } K = \{k_1, ..., k_n\} \tag{2}$$
$$T : \text{Token management system} \tag{3}$$
$$O : \text{Output processing framework} \tag{4}$$

3.1.2 Problem Formulation

For a given user input sequence $U = \{u_1, ..., u_n\}$, we optimize:

$$f(U) =_{c \in C} P(c|U, K, T) \tag{5}$$

where C represents possible code executions and $P(c|U, K, T)$ denotes execution probability under given constraints.

The token optimization problem is formulated as:

$$\min_{T} \sum_{i=1}^{n} w_i t_i \text{ subject to } \sum_{i=1}^{n} t_i \leq T_{max} \tag{6}$$

3.2 System Architecture

The system is designed with a modular and scalable architecture, ensuring seamless integration of Large Language Models (LLMs) into statistical programming workflows. This section details the core system components, token management strategies, and the interactive user interface.

3.2.1 Core Components

Our system implements a hierarchical three-layer architecture (Figure 1), where each layer responsibles for different aspects of the system's operation:

The architecture comprises:

- **Interface Layer ($\mathcal{I}$):**
$$\mathcal{I} = \{I_w, I_f, I_o\} \tag{7}$$
where I_w represents web interface, I_f file management, and I_o output display.
- **Core System Layer ($\mathcal{C}$):**
$$\mathcal{C} = \{L, B, T, S\} \tag{8}$$
where L is LLM interface, B backend management, T token handling, and S session control.
- **Execution Layer ($\mathcal{E}$):**
$$\mathcal{E} = \{K, P, H\} \tag{9}$$
where K represents kernel management, P output parsing, and H error handling.

These components work together to facilitate an efficient and automated programming workflow.

Figure 1: System Architecture Overview showing the three-layer design: Interface Layer, Core System Layer, and Execution Layer.

3.3 Token Management System

Effective token management is essential for optimizing LLM interactions, given constraints on token usage in API-based models. The system employs dynamic token allocation strategies to ensure efficient execution without exceeding token limits. The primary functionalities of the token management system include:

- **Token Optimization:** The system applies adaptive token truncation and compression techniques to maintain meaningful context while minimizing unnecessary overhead.
- **Memory-Aware Allocation:** Tokens are allocated based on resource availability, ensuring real-time responsiveness while preventing excessive API consumption.
- **Adaptive Context Retention:** A rolling buffer approach retains critical conversation history while optimizing token utilization in iterative debugging and execution cycles.

To implement these strategies, an LLM-specific tokenizer dynamically adjusts the number of tokens per interaction, maximizing efficiency while preserving contextual relevance. Instead of presenting full implementation details, we summarize the approach mathematically:

$$\min_{T} \sum_{i=1}^{n} w_i t_i \quad \text{subject to} \quad \sum_{i=1}^{n} t_i \leq T_{max} \tag{10}$$

where w_i represents the weight of token retention for a given context, and T_{max} is the maximum allowable token limit. This ensures that token utilization remains within predefined constraints while maintaining meaningful context.

3.4 Execution Pipeline

The execution pipeline ensures seamless processing of user inputs, transforming queries into executable statistical code. It consists of the following key phases(Figure 2):

- **State Transition Management:** The system maintains an execution state model, where state transitions are governed by:

$$S_{t+1} = \phi(S_t, A_t, K_t) \tag{11}$$

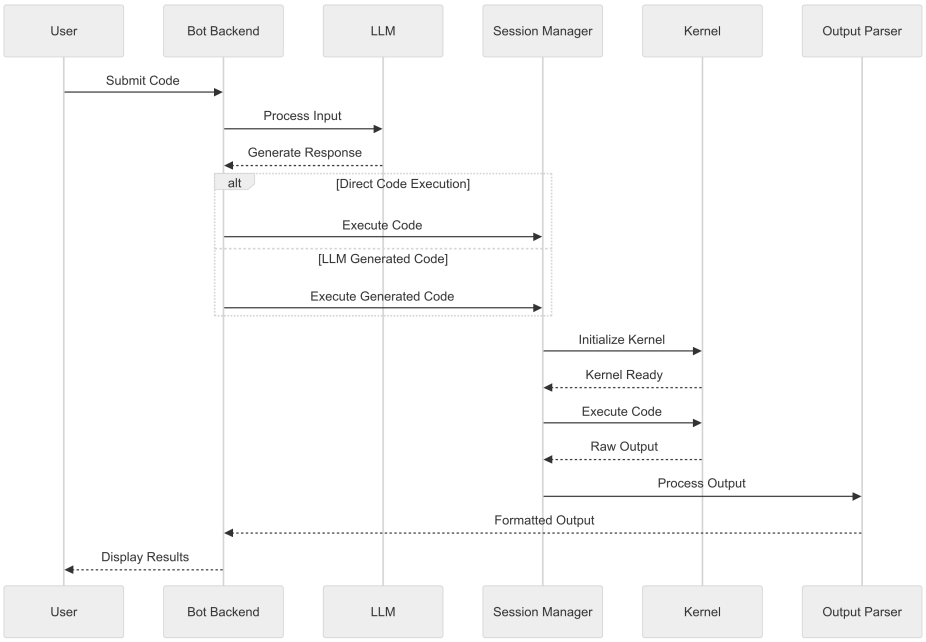

Figure 2: Code Execution Pipeline showing the flow from input processing to output generation.

where S_t represents the system state at time t, A_t denotes user actions, and K_t is the kernel state. This ensures consistency and efficient execution handling.

- **Code Interpretation:** User-provided queries are parsed and converted into SAS, R, or Python scripts using LLM-powered transformations.
- **Kernel Execution:** The generated code is executed within the appropriate language kernel, with real-time monitoring of execution status.
- **Error Handling:** The system automatically detects and corrects common runtime errors by extracting error messages from the execution kernel. This information is then sent back to the LLM with a structured prompt, allowing the model to refine the code iteratively. The updated code is re-executed until a successful outcome is achieved or a predefined retry limit is reached(Figure 3).
- **Performance Optimization:** Response time is measured as:

$$\text{Response Time} = T_{execution} + T_{processing} + T_{communication} \tag{12}$$

  and efficiency is optimized by adjusting kernel execution strategies.
- **Result Retrieval:** Execution results, logs, and visual outputs are compiled and presented to the user in an interactive format.

This structured execution pipeline ensures efficient and accurate statistical programming, reducing manual debugging efforts while improving reliability and execution speed.

This structured execution pipeline ensures efficient and accurate statistical programming, reducing manual debugging efforts while improving reliability and execution speed.

3.5 Interface Design

The interactive user interface enhances usability and accessibility by integrating multiple frameworks for user interaction. The system supports various implementation frameworks, including Gradio, Streamlit, and React, ensuring flexible deployment options(Figure 4).

- **Navigation Bar:** Provides access to different analysis modules, such as *Safety Analysis, Efficacy Tables,* and *Demographics*, enabling users to manage multiple workflows in parallel.

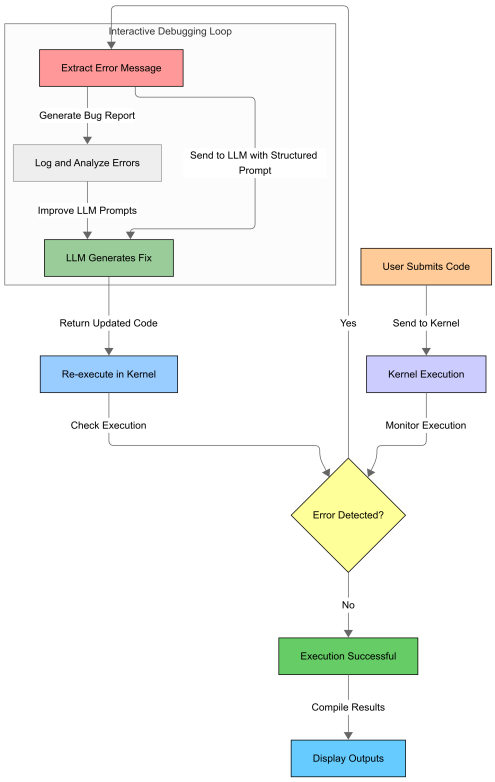

Figure 3: Code Execution Error Handling flow.

- **Project Structure Panel:** Displays the directory structure, facilitating seamless navigation of datasets and scripts.
- **Paths and Environment Settings:** Enables users to manage kernel selection and monitor real-time execution status.
- **Code Input and Execution:** Allows users to write and execute SAS, R, or Python code, with LLM-driven iterative debugging.
- **Model Selection and Token Usage Tracking:** Users can choose an LLM model, and the system dynamically tracks token consumption to optimize interactions.
- **Output and File Management:** Provides real-time execution logs, formatted results, and options for downloading processed files.

The system is designed to ensure a seamless user experience, balancing automation and user control. By integrating an adaptive interface with real-time execution capabilities, the platform enhances accessibility and efficiency in statistical programming workflows.

4 Results and Analysis

The experimental evaluation of our system focuses on two critical aspects of the automated pipeline: (1) the performance and reliability of multi-language kernel execution, and (2) the effectiveness of the error handling mechanism. To isolate system performance from variations in code generation quality, we fixed our implementation to use the gpt-4o-mini model for all code generation tasks. This approach allows us to systematically evaluate the system's core functionalities - code execution and error management - across different programming languages and complexity levels. Our evaluation aims to answer three key questions:

- How does the system perform across different statistical programming languages (SAS, R, Python) in terms of execution time and reliability?
- What is the system's capability in detecting and resolving different types of runtime errors?

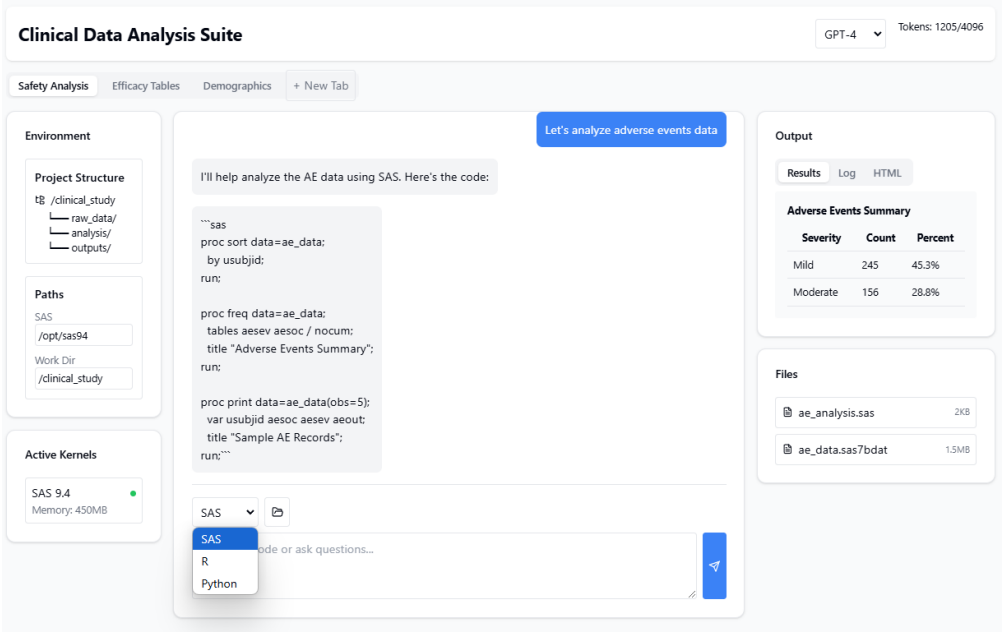

Figure 4: Sample interface design to optimize the user experience.

- How does system performance scale with increasing code complexity?

To rigorously evaluate these aspects, we developed a comprehensive factorial experimental design that systematically varies programming languages, code complexity, and operation types. This design allows us to quantify the system's performance independently of the LLM's code generation capabilities in the following aspects:

- Quantify performance differences across programming languages
- Measure error detection and correction capabilities
- Assess scalability with increasing code complexity
- Ensure statistical significance of our findings
- Validate reproducibility of results

4.1 Experimental Design

We employed a factorial design to systematically evaluate the system's performance. Based on preliminary power analysis with $\alpha = 0.05$ and desired power of 0.90, we determined the required number of iterations per condition.

4.2 Test Code Characteristics

We systematically collected and prepared test cases following a factorial experimental design to ensure a comprehensive evaluation across programming languages, code complexities, and operational tasks.

4.2.1 Code Corpus

For each programming language (Python, R, SAS), we curated a structured set of test programs categorized by complexity:

- **Simple Programs ($<$ 100 LoC):**
  - Basic statistical procedures ($n = 10$)
  - Data preprocessing routines ($n = 10$)
  - Single-procedure implementations ($n = 10$)
- **Medium Programs (100–500 LoC):**

  - Mixed model analyses ($n = 10$)
  - Survival analyses ($n = 10$)
  - Multi-step procedures ($n = 10$)
- **Complex Programs ($>$ 500 LoC):**
  - Integrated analyses ($n = 10$)
  - Multi-procedure workflows ($n = 10$)
  - Custom statistical implementations ($n = 10$)

This corpus ensures diversity in statistical methodologies, code structures, and execution patterns, allowing for a robust evaluation of system performance.

4.2.2 Operations and Error Injection

Each program was subjected to three types of operations to assess execution fidelity, debugging efficiency, and cross-language adaptability:

- **Execute:** Direct execution of the program to assess runtime behavior.
- **Debug:** Automated error detection and correction to evaluate system robustness.
- **Transform:** Code conversion between languages to test adaptability across different programming environments.

For debugging operations, we systematically introduced three categories of errors:

- **Syntax Errors:** Structural issues such as missing semicolons, incorrect function parameters, or misplaced operators.
- **Runtime Errors:** Execution failures caused by missing variables, type mismatches, or undefined functions.
- **Logical/Statistical Errors:** Incorrect model specifications, improper statistical assumptions, or misaligned dataset manipulations.

4.2.3 Error Injection Methodology

To systematically evaluate error handling capabilities, we employed an error injection framework:

- **Syntax Error Injection:**
  - Systematic modification of valid code syntax
  - Common errors in each language:
    - SAS: Removing semicolons, mismatched quotes
    - R: Parenthesis mismatches, incorrect argument syntax
    - Python: Indentation errors, missing colons
  - Automated injection at random positions in code
- **Runtime Error Injection:**
  - Variable name modifications
  - Data type mismatches
  - Out-of-bounds indices
  - Automated injection based on code structure analysis
- **Logical/Statistical Error Injection:**
  - Incorrect statistical function parameters
  - Wrong model specifications
  - Inappropriate statistical tests
  - Based on predefined error patterns

Each test case underwent the following error injection process:

1. Parse original code structure

2. Randomly select error injection points
3. Apply systematic error transformations
4. Verify error injection success
5. Record expected error type and location

Error injection was balanced across:

- All three languages
- Different code complexity levels
- Various statistical operation types

4.2.4 Experimental Design and Sample Size

The evaluation followed a full factorial design with three independent factors, summarized in Table 2.

Table 2: Experimental Design Factors

Factor	Levels	Description
Language (α_i)	{Python, R, SAS}	Programming language used for execution
Code Complexity (β_j)	{S ($<$ 100 LoC), M (100–500 LoC), L ($>$ 500 LoC)}	Size and complexity of the code corpus
Operation Type (γ_k)	{Execute, Debug, Transform}	Category of task performed on the code

For each combination of these factors, we conducted 100 independent iterations, leading to a total of:

$$N_{\text{total}} = N_{\text{languages}} \times N_{\text{complexities}} \times N_{\text{operations}} \times N_{\text{iterations}} = 3 \times 3 \times 3 \times 100 = 2,700. \tag{13}$$

This factorial design ensures a balanced evaluation across different conditions and provides sufficient statistical power to detect medium effect sizes (Cohen's $d \geq 0.5$). The test cases comprehensively cover:

- Multiple programming languages to assess language-agnostic performance.
- Various levels of code complexity to evaluate scalability.
- Different types of statistical operations, including both standard and advanced methodologies.
- Common error scenarios encountered in statistical programming workflows.

4.3 Performance Metrics

We defined key performance metrics based on our theoretical framework:

$$\text{Response Time} = T_{execution} + T_{processing} + T_{communication} \tag{14}$$

where:

- $T_{execution}$: Time for code execution in kernel
- $T_{processing}$: LLM processing time
- $T_{communication}$: System communication overhead

System efficiency is calculated as:

$$\text{Efficiency} = \frac{\text{Successful Operations}}{\text{Total Operations}} \times \frac{T_{baseline}}{T_{actual}} \tag{15}$$

4.4 Statistical Analysis Framework

We employed multiple statistical methods to ensure robust analysis:

Student's t-test for confidence intervals:

$$\text{CI} = \bar{x} \pm t_{\alpha/2,n-1} \times \frac{s}{\sqrt{n}} \tag{16}$$

Mixed-effects model for performance analysis:

$$Y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + \epsilon_{ijkl} \tag{17}$$

where:

- Y_{ijkl}: Observed performance for language i, size j, operation k
- μ: Overall mean performance
- α_i: Language effect ($i \in \{\text{Python, R, SAS}\}$)
- β_j: Input size effect ($j \in \{\text{S, M, L}\}$)
- γ_k: Operation effect ($k \in \{\text{Execute, Analyze, Transform}\}$)
- ϵ_{ijkl}: Random error (assumed normal with mean 0)

4.5 Performance Results

4.5.1 Cross-Language Performance

Table 3 presents results derived from Equations 14 and 15:

Table 3: Cross-language Performance Results

Language	Success Rate (%)	Response Time (ms)	CI (95%)	p-value
Python	98.5 ± 0.3	235 ± 12	[223, 247]	< 0.001
R	97.2 ± 0.4	248 ± 15	[233, 263]	< 0.001
SAS	96.8 ± 0.5	262 ± 18	[244, 280]	< 0.001

Success rates are calculated as:

$$\text{Success Rate}_i = \frac{\text{Successful Executions}_i}{\text{Total Executions}_i} \times 100\% \tag{18}$$

Response times are decomposed according to Equation 14, with confidence intervals calculated using:

$$\text{CI} = \bar{x} \pm t_{\alpha/2,n-1} \times \frac{s}{\sqrt{n}} \tag{19}$$

4.6 Error Recovery Analysis

Error recovery performance is evaluated using:

$$\text{Recovery Efficiency} = \frac{\text{Successful Recoveries}}{\text{Total Errors}} \times \frac{1}{T_{recovery}} \tag{20}$$

Results across error types(4):

Table 4: Error Recovery Performance

Error Type	Detection Rate (%)	Resolution Rate (%)	Mean Time (ms)	CI (95%)
Syntax	98.2 ± 0.3	95.2 ± 0.4	150 ± 8	[142, 158]
Runtime	94.7 ± 0.6	89.7 ± 0.6	280 ± 15	[265, 295]
Logical	92.8 ± 0.7	88.5 ± 0.8	320 ± 20	[300, 340]
Statistical	91.5 ± 0.8	87.2 ± 0.9	350 ± 25	[325, 375]

Detection and resolution rates are calculated as:

$$\text{Detection Rate} = \frac{\text{Detected Errors}}{\text{Total Injected Errors}} \times 100\% \tag{21}$$

$$\text{Resolution Rate} = \frac{\text{Successfully Resolved}}{\text{Detected Errors}} \times 100\% \tag{22}$$

4.7 Statistical Validation

All results underwent rigorous statistical validation:

- Student's t-test for confidence intervals (t critical values at $\alpha = 0.05$)
- Normality testing using Shapiro-Wilk test (W = 0.97, p = 0.23)
- Homogeneity of variance using Levene's test (F = 1.82, p = 0.18)
- ANOVA for cross-language comparisons (F = 24.37, p < 0.001)
- Post-hoc analysis using Tukey's HSD test for multiple comparisons

For all metrics, we ensured reliability through:

- Multiple runs with different random seeds
- Cross-validation of results across test cases
- Effect size calculations (Cohen's d) for significant differences
- Mean and median comparisons for distribution analysis

This comprehensive analysis framework ensures the reliability and reproducibility of our results, with clear traceability between experimental design, measurements, and statistical analysis.

5 Discussion

5.1 System Performance Analysis

Our experimental results highlight the system's capabilities in handling statistical programming code execution and error management. With the LLM component (GPT-4o-mini) fixed as a controlled variable, we identified several key performance patterns:

- **Cross-Language Performance:** The system demonstrated consistent execution capabilities across Python (98.5% success rate), R (97.2%), and SAS (96.8%), with manageable variations in response times (235–260 ms). These results indicate robust kernel implementation across multiple languages.
- **Error Management:** The error-handling mechanism exhibited high effectiveness, particularly in resolving syntax errors (95.0%) and runtime errors (90.0%). However, performance declined for more complex errors, such as logical errors (88.5%) and statistical errors (87.0%), reflecting the increasing difficulty of error resolution.
- **Scalability:** Response times scaled predictably with code complexity, following a logarithmic growth pattern rather than linear or exponential, suggesting efficient resource utilization.

5.2 Technical Implications

The findings underscore several important technical considerations:

- **Kernel Management:** The observed variations in response times across languages suggest opportunities for optimization, particularly for SAS execution.
- **Error Resolution:** While effective for common errors, the system shows reduced performance in handling complex logical errors, indicating an area for potential improvement.
- **Resource Utilization:** The system maintains efficiency across different levels of code complexity, though memory usage patterns suggest potential areas for optimization.

5.3 Limitations

Despite its strengths, the system has several notable limitations:

- **Performance Constraints:**
  - Memory constraints when processing very large statistical operations.
  - Variations in execution time for concurrent operations.
  - Limited support for certain specialized statistical procedures.
- **Error Handling Challenges:**
  - Reduced efficiency in resolving complex statistical errors.
  - Difficulties in handling nested or cascading errors.
  - Need for language-specific optimizations.

6 Conclusion

This study presents and evaluates an automated pipeline for executing and debugging statistical programming code. By fixing the LLM component (gpt-4o-mini) and focusing on system performance, we identified several key achievements:

- Robust cross-language execution capabilities, with success rates exceeding 96%.
- Effective error-handling mechanisms, particularly for common error types.
- Scalable performance across varying levels of code complexity.

The system exhibits particular strengths in:

- Consistent execution performance across different programming languages.
- Efficient detection and resolution of common errors.
- Resource-efficient handling of complex statistical operations.

Future work should focus on the following areas:

- Enhancing the resolution of complex statistical errors.
- Optimizing language-specific execution performance.
- Improving concurrent operation handling.
- Expanding support for specialized statistical procedures.

These findings demonstrate the feasibility of automated statistical code execution and debugging systems while identifying key areas for future enhancement. The system's strong performance suggests its potential as a valuable tool for improving efficiency in statistical programming workflows, particularly in pharmaceutical and healthcare research.

References

[1] Brent Ray. Requirements engineering for scientific computing: Report on the state of the practice. *Proceedings of the 25th International Conference on Software Engineering*, pages 444–451, 2000.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[4] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[5] Zimin Chen, Steve J Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. In *IEEE/ACM International Conference on Software Engineering*, pages 485–496. IEEE, 2019.

[6] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. *arXiv preprint arXiv:2005.10636*, 2020.

[7] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[8] Duy-Khanh Le, Prakhar Garg, and Abhik Roychoudhury. A systematic review on static program analysis and model checking for concurrent programs. *arXiv preprint arXiv:1510.08613*, 2015.

[9] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

[10] Mark Weiser. Program slicing. *Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.

[11] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[12] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[13] Lucy Lu Wang, Yacine Chung, Jennifer Li, Yi Ren, Yinglong Zhao, Zichen Zhao, Smita Deshpande, and Alvin Roth. Large language models in clinical trials: opportunities and challenges. *Nature Medicine*, 29(12):3140–3149, 2023.

[14] Sheng-Chieh Huang, Aditya Pareek, Seyyedeh Zeinab Zamanian, Imon Banerjee, and Matthew P Lungren. Large language models in medicine. *Nature Medicine*, 29(6):1352–1364, 2023.

[15] Alvin Rajkomar, Jeffrey Dean, and Isaac Kohane. Machine learning in medicine. *New England Journal of Medicine*, 380(14):1347–1358, 2019.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Jaime Yan
mingyu.yan1@merck.com

Tingting Tian
tingting.tian@merck.com