An End-to-End Approach to Fine-Tune Small LLMs for Generating Admiral R Code in Statistical Programming

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

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

Abstract

This paper presents a method for fine-tuning the LLaMA 3.1 8B model using Low-Rank Adaptation (LoRA) to generate R code with the Admiral package for ADaM dataset creation. The process begins by querying a large language model (LLM) with structured JSON specifications to extract relevant knowledge, generating question-answer pairs with Admiral-based R code. EVLO instruct tuning[1] and rule-based augmentation enhance data quality, followed by validation using a knowledge graph-based query system that filters records with a confidence threshold of 0.8. The curated dataset is then used to fine-tune LLaMA 3.1 with Unsloth's[2] PEFT framework, enabling it to generate Admiral R code based on structured ADaM specifications, including variable metadata and derivation logic. Model performance is evaluated using the knowledge graph to assess answer confidence. While the generated code requires user validation, this approach establishes a systematic method for fine-tuning LLMs to automate and enhance ADaM dataset creation in statistical programming.

***K*eywords** LLMs, Admiral R, ADaM, LoRA, Statistical Programming, Clinical Data, Code Generation

1 Introduction

The creation of *Analysis Data Model (ADaM)* datasets in clinical trials is a complex, time-consuming, and error-prone process that requires strict adherence to *Clinical Data Interchange Standards Consortium (CDISC)* guidelines. Statistical programmers often rely on packages such as *Admiral*[1] to streamline this process. However, despite Admiral's structured framework, ADaM dataset generation still involves substantial manual effort, particularly for complex datasets with intricate derivation logic.

1.1 Challenges in Automating ADaM Dataset Creation

Recent advancements in *Large Language Models (LLMs)*, such as *Claude 3.5* and *GPT-4o*, have enabled the generation of Admiral-based R code directly from user queries. Since Admiral is a publicly available R package with extensive documentation online (e.g., GitHub, rdrr.io), modern LLMs have been trained on this data, allowing them to generate ADaM-related code. However, despite these advancements, three fundamental challenges persist:

1. *Validation of Generated Code*: Ensuring that LLM-generated code is correct and fully adheres to ADaM specifications is nontrivial. While LLMs can generate code snippets, there is no built-in mechanism to verify correctness, increasing the risk of compliance issues.
2. *Prompt Engineering and Automation*: Users must carefully craft complex prompts and queries to generate meaningful and structured results, making it impractical for large-scale ADaM dataset creation. Scaling this process requires either an *API-driven pipeline* for batch queries or labor-intensive manual interactions.
3. *Data Privacy Concerns*: Using cloud-based LLM APIs introduces potential risks of exposing sensitive clinical trial data. Many organizations require *self-hosted* solutions to comply with *data security* and *regulatory* constraints.

[1] `https://pharmaverse.github.io/admiral/`

1.2 Proposed Hybrid Approach

To address these challenges, this paper presents a *hybrid approach* that enables the automatic generation of *Admiral-based ADaM datasets* while offering two deployment options:

- *Cloud-Based Workflow* (for users without privacy concerns): We introduce a *pipeline* that converts *Excel-based ADaM specifications* into a structured *JSON format*. This JSON representation is then used within a *knowledge graph query system* and a structured *prompting framework* to automatically batch-generate Admiral code using cloud-based LLMs.
- *Local Fine-Tuned LLM Workflow* (for users requiring data security): We fine-tune *LLaMA 3.1 8B* using *Low-Rank Adaptation (LoRA)* and domain-specific knowledge to generate Admiral R code locally. While not as powerful as state-of-the-art models like *GPT-4o*, our fine-tuned model achieves *strong performance* in generating ADaM-compliant code without exposing any data externally.

1.3 Role of Knowledge Graphs in Querying, Validation, and Fine-Tuning

A *knowledge graph* plays a crucial role in both the *query system* and *fine-tuning data preparation*, addressing multiple challenges:

1. *Automated Query-Based Code Generation*: The knowledge graph enhances the structured prompting pipeline by linking *ADaM specification elements* to *Admiral functions and CDISC guidelines*, ensuring that the generated code aligns with domain standards.
2. *Fine-Tuning Data Preparation*: The knowledge graph assists in filtering, structuring, and augmenting training data for fine-tuning the *LLaMA 3.1 8B* model, improving the contextual accuracy of generated outputs.
3. *Validation of Generated Code*: A knowledge graph-based evaluation system assigns *confidence scores* to generated code snippets, filtering out unreliable results. This ensures compliance with ADaM requirements and reduces manual review efforts.

1.4 Evaluation and Performance Assessment

To systematically evaluate our approach, we introduce the *Overall Performance Score (OPS) metric*, which assesses the generated code across multiple dimensions:

- *Syntax and Functional Accuracy*: Ensures that the generated code is syntactically correct and executable.
- *Adherence to CDISC and ADaM Standards*: Measures compliance with industry standards.
- *Code Efficiency and Readability*: Evaluates maintainability and best-practice adherence.

Experimental results demonstrate that our fine-tuned *LLaMA 3.1 8B model* provides a reliable alternative to cloud-based solutions while preserving data security. Although cloud-based LLMs such as *Claude 3.5* and *GPT-4o* generate higher-quality code, our *local model achieves competitive performance*, making it a viable solution for privacy-sensitive environments.

1.5 Why Our Proposed Pipeline Works

Our proposed pipeline is designed to seamlessly integrate with Admiral's existing *function-based, modular approach*, which processes **one variable at a time** before merging results into the final dataset. This inherent characteristic of Admiral makes our pipeline naturally well-suited for automating the **generation of question-answer pairs**, where:

- The *question* is derived from ADaM specifications (structured metadata for a variable).
- The *answer* is the corresponding *Admiral R code* that performs the derivation.

This approach enables efficient **training material generation** and aligns with existing **LLM response formats**, allowing for seamless automation.

1.5.1 Step-by-Step Process

1. *Extract Variable Information from ADaM Specification:* Each ADaM dataset variable is processed independently, making it easy to structure into a *question-answer pair* format.

2. *Generate Admiral Code for Each Variable:* Using structured prompts or a fine-tuned LLM, we automatically generate Admiral R code corresponding to the variable's derivation logic.
3. *Post-Processing and Dataset Combination:* Once all variables are processed, the individual code snippets are merged to construct the final ADaM dataset.
4. *Validation via Knowledge Graph Query System:* A structured validation mechanism ensures correctness by checking the generated code against predefined derivation rules in a knowledge graph.

1.5.2 Generalizability to Other Coding Packages

The proposed pipeline is *not limited to Admiral* but is applicable to any *statistical programming package* that follows a similar *function-based, modular approach*. This includes:

- *Other CDISC-compliant R packages* for ADaM dataset creation.
- *SAS-based pipelines* where macro functions generate individual dataset variables.
- *Python-based solutions* that adopt a stepwise transformation process.

By structuring ADaM dataset creation as a *modular, function-based pipeline*, we ensure that our approach is both *scalable* and *adaptable* across multiple coding environments.

1.6 Contributions of This Work

This paper makes the following key contributions:

1. *Automated ADaM Specification Processing*: We introduce a *structured pipeline* that converts ADaM dataset specifications into *machine-readable JSON*, enabling batch processing for automatic code generation.
2. *Hybrid LLM-Based Code Generation*: We propose a method that integrates both *cloud-based LLMs* and a *locally fine-tuned LLaMA model*, providing flexibility based on data privacy requirements.
3. *Systematic Code Validation Framework*: We introduce a *knowledge graph-based validation system* that assigns confidence scores to generated code, improving reliability and reducing manual review efforts.
4. *OPS-Based Evaluation Metric*: We develop an *Overall Performance Score (OPS)* to assess generated code across multiple quality dimensions.

1.7 Paper Organization

The remainder of this paper is structured as follows:

- Section 2 provides a review of related work in LLM-based code generation and knowledge graph applications in statistical programming.
- Section 3 describes our methodology, including dataset preparation, fine-tuning techniques, and structured query design.
- Section 4 and Section 5presents experimental results, comparing our fine-tuned model with existing cloud-based LLM solutions.
- Section 6 and Section 7 discusses the implications of our findings, limitations, and potential future research directions.
- Finally, Section 8 concludes the paper with a summary of key contributions.

By integrating *structured knowledge extraction*, *fine-tuned LLMs*, and *automated validation*, this work establishes a scalable method for enhancing *statistical programming workflows* in clinical trials.

2 Related Work

The use of large language models (LLMs) for code generation has gained significant attention in recent years. Models like Codex [3] and AlphaCode [4] have demonstrated impressive capabilities in generating general-purpose programming code. However, their application to domain-specific tasks, such as statistical programming in clinical trials, remains underexplored.

Several studies have explored fine-tuning LLMs for specialized tasks. For instance, LoRA [5] has emerged as an effective technique for adapting pre-trained models to new domains with minimal computational overhead. Similarly, domain-specific adaptations of LLMs have been successfully applied in areas such as legal document analysis [6] and biomedical text generation [7] as well as sas code generation [8]. Our work extends these approaches by applying LoRA-based fine-tuning to generate R code for ADaM dataset creation.

Knowledge graph integration has also been explored as a means of enhancing LLM outputs. For example, Wang et al. [9] proposed incorporating structured knowledge into language models to improve task-specific performance. In our methodology, we leverage a knowledge graph derived from Admiral documentation to validate generated code and ensure compliance with CDISC standards.

Finally, prior research on automated testing frameworks for generated code [10] highlights the importance of execution accuracy and structural consistency in evaluating code quality. Our proposed evaluation framework incorporates these metrics alongside domain-specific considerations, providing a comprehensive assessment of generated code.

By combining these advancements, our work addresses the unique challenges of automating statistical programming workflows while ensuring high-quality and compliant code generation.

3 Methodology

3.1 Overview

This research aims to address the core challenges in automating ADaM dataset generation using the *Admiral* package. Specifically, we focus on three key problems:

1. *Automating the Admiral Code Generation Process*: We design a structured pipeline that converts ADaM variable specifications from Excel-based formats into a structured *question-answer* format. The process involves:
   - Extracting ADaM variable metadata from the specification file.
   - Structuring the metadata as the *question* part of a JSON-based dataset.
   - Defining a standardized output format to serve as the *answer*, ensuring compatibility with LLM responses.
   - Using a Python script to generate structured JSON files containing the question-answer pairs.
   - Feeding these structured queries to an LLM, obtaining responses that generate Admiral R code per variable.
   - Combining the individual code snippets, applying post-processing, and assembling the final ADaM dataset code.
2. *Validating LLM-Generated Admiral Code*: Ensuring the correctness and quality of LLM-generated code is essential for regulatory compliance and practical usability. We address this challenge by:
   - Crawling official Admiral documentation from `pharmaverse.github.io` to collect function specifications, usage examples, and rules.
   - Constructing a *knowledge graph* based on extracted documentation to capture relationships between Admiral functions and ADaM dataset rules.
   - Using the knowledge graph as a validator to systematically assess the quality and correctness of generated Admiral code.
3. *Developing a Local Processing Alternative*: For organizations requiring an offline solution due to *data security concerns*, we develop a local pipeline that integrates:
   - A fine-tuned lightweight model, optimized using the `Unsloth PEFT` framework, to generate Admiral code locally.
   - The knowledge graph-based query and validation system to enhance reliability.

To achieve these objectives, the following subsections will detail:

- Data processing techniques for extracting and structuring ADaM specifications.
- Document processing methods for constructing the knowledge graph.
- The architecture of the query and validation pipeline.
- The fine-tuning process, including training data preparation and optimization strategies.

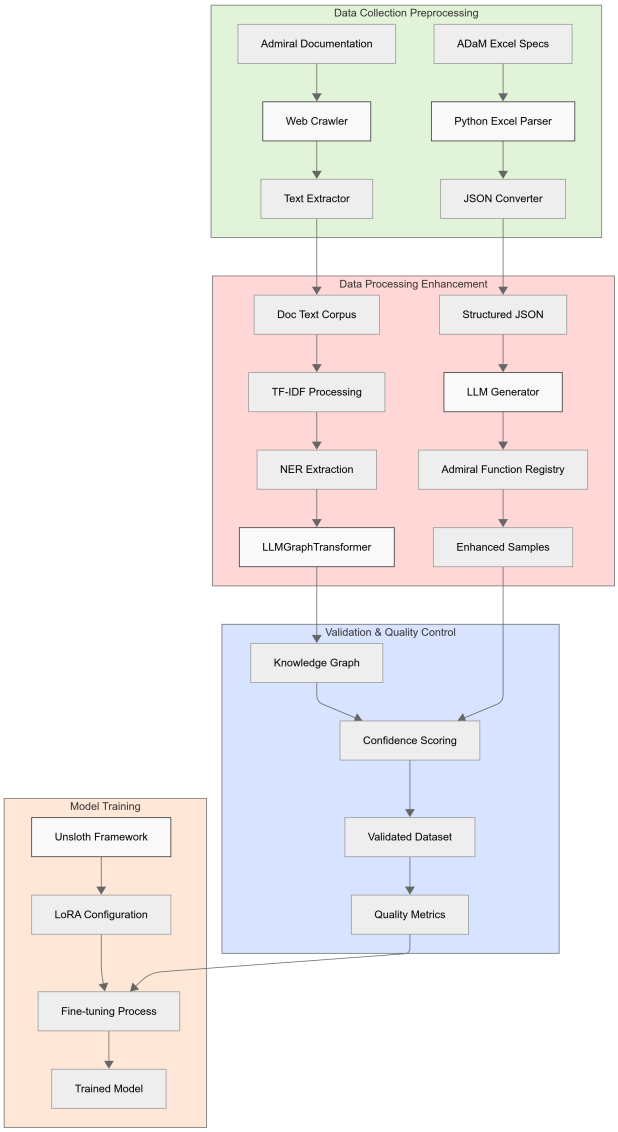

Figure 1: Overall System Architecture showing the integration of data processing, knowledge graph construction, and model training components.

3.1.1 Pipeline Architecture

As illustrated in Figure 1, our pipeline implementation follows a modular architecture with clearly defined interfaces between components. Each stage is designed to be independently verifiable and optimizable, allowing for iterative improvements without affecting the entire pipeline.

The key innovations in our approach include:

1. **Structured Data Transformation**: A novel Excel-to-JSON conversion process that preserves the hierarchical relationships inherent in ADaM specifications while maintaining CDISC compliance.

2. **Dual Enhancement Strategy**: Combining function-based generation with evolution-based augmentation to create diverse, yet valid training samples.

3. **Knowledge Graph Validation**: Implementation of a specialized LLMGraphTransformer that ensures generated code maintains semantic consistency with Admiral package documentation.

4. **Optimized Fine-tuning**: Careful configuration of LoRA parameters and training procedures to maximize model performance while minimizing computational requirements.

3.2 Data Collection and Preprocessing

The data collection process implements a systematic approach to gathering and structuring information from multiple sources. The primary data sources include ADaM specifications in Excel format and Admiral package documentation. Each source requires specific processing techniques to ensure data quality and consistency.

3.2.1 ADaM Specification Processing

The processing of ADaM specifications follows a structured pipeline to ensure data integrity and CDISC compliance. A custom Excel parser was developed to handle three critical components:

1. **Content Sheet Processing:** This component extracts dataset-level metadata and structural relationships, preserving the hierarchical nature of ADaM specifications. The parser identifies key variables, their relationships, and associated metadata while adhering to CDISC naming conventions.
2. **Codelist Integration:** The system processes controlled terminology mappings, creating a comprehensive reference system for variable values. This ensures consistency in terminology usage and maintains compliance with CDISC standards.
3. **Dataset Structure Analysis:** Individual dataset specifications undergo detailed analysis to identify variable relationships, dependencies, and implementation requirements. This process creates a structured representation of the complex relationships present in clinical trial data.

3.2.2 Extracted JSON Structure from Specification

The ADaM specification is initially extracted into a hierarchical JSON structure that captures the essential elements of dataset organization. This structure encompasses dataset-level attributes, including its structural type and key variables, followed by a detailed array of variable definitions. Each variable entry contains core metadata such as variable name, label, data type, associated codelists, and implementation specifications. This standardized format serves as the foundation for subsequent transformation into our question-answer training format.

```
{
  "Dataset Name": {
    "Structure": "str",
    "Key Variables": "str",
    "Variables": [
      {
        "VARIABLE": "str",
        "LABEL": "str",
        "DATA TYPE": "str",
        "Codelist": "NoneType",
        "Code": "str",
        "Terms": [
          "..."
        ]
      }
    ]
  }
}
```

3.2.3 Structured Question-Answer Format Design

To facilitate effective model training and ensure consistent code generation, we developed a specialized JSON-based format that structures ADaM specifications into question-answer pairs. This format bridges the gap between traditional ADaM specifications and LLM training requirements while maintaining traceability to source specifications.

Question Format Structure The question component follows a standardized template:

```
{
    "Question": "Dataset identification: [dataset name]\n
                Variable: [variable name]\n
                Specification: [variable specification]"
}
```

This structure was designed to:

- Maintain clear dataset context and variable relationships
- Preserve essential derivation requirements
- Present specifications in a consistent, machine-readable format

Answer Format Design The answer component implements a structured format optimized for Admiral code generation:

```
{
    "Answer": "<<admiral_functions: [required functions]>>\n
               <<source_datasets: [required datasets]>>\n
               <<indexes: [required index variables]>>\n
               ####\n
               [implementation code]"
}
```

This format provides several advantages:

- Explicit identification of required Admiral functions
- Clear specification of dataset dependencies
- Separation of metadata from implementation code
- Structured framework for validation and testing

Format Conversion Process The conversion from traditional ADaM specifications to our structured format involves:

1. Extraction of relevant specification components from source JSON
2. Template-based question formation using extracted metadata
3. LLM-assisted answer generation with custom prompting
4. Validation of generated pairs against knowledge graph patterns

This structured approach ensures consistency in training data while maintaining the semantic richness of original ADaM specifications. The format's design facilitates both automated processing and human review, supporting our dual objectives of efficiency and accuracy in code generation.

3.2.4 Admiral Tech Documentation Processing

The Admiral package documentation undergoes a specialized extraction and processing pipeline. This process involves:

1. **Web Crawling:** A custom crawler extracts documentation from the pharmaverse.github.io repository, focusing on function specifications, usage patterns, and implementation examples.
2. **Text Processing:** Advanced natural language processing (NLP) techniques are applied to identify key concepts, recognize code patterns, and determine relationships between different package components.
3. **Code Block Analysis:** The system extracts and analyzes code examples, identifying common patterns and best practices in Admiral function usage.

3.3 Data Enhancement and Augmentation

Our data enhancement approach implements a dual-strategy system (Figure 2) that combines function-based generation with evolution-based[1] enhancement. This comprehensive approach ensures both breadth and depth in the training dataset while maintaining strict adherence to CDISC standards and Admiral package specifications.

1. **Evolution-Based Enhancement[1]:** This strategy enhances existing specifications, drawing inspiration from EVLO Instruct Tuning[1]. It focuses on scaling complexity, adding variable relationships and dependencies, enhancing implementation patterns, incorporating error handling and validation, adapting generation patterns based on complexity analysis, implementing evolution-based instruction refinement, and incorporating domain-specific constraints and best practices.
2. **Function-Based Generation:** This strategy systematically creates new training samples, similar to the described function-based augmentation approach. It involves analyzing Admiral function signatures and documentation, generating template specifications for each function, creating variations through parameter combination, validating against known usage patterns, applying predefined transformation rules to existing samples, ensuring coverage of edge cases and error conditions, and maintaining CDISC compliance throughout variations.

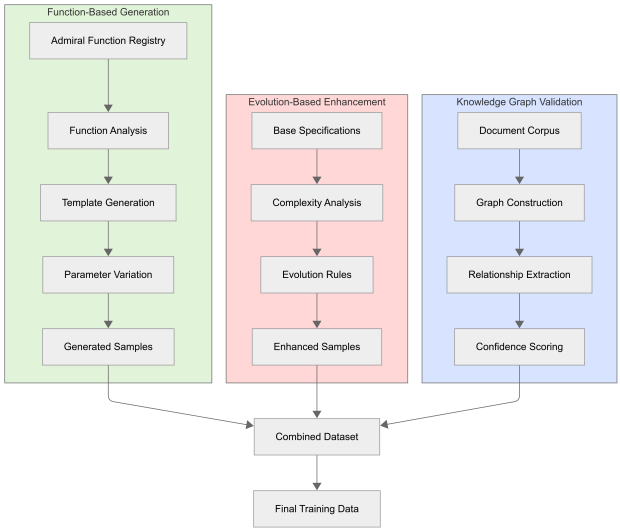

Figure 2: Dual Enhancement Strategy showing EVLO tuning and rule-based augmentation, while use knowledge graph as the validator

3.4 Knowledge Graph Construction

The knowledge graph is a central component of our methodology, providing a structured representation of relationships between ADaM specifications, Admiral functions, and implementation patterns. This graph not only validates generated code but also guides the generation process itself. Figure 3 shows the knowledge graph building flow.

3.4.1 Graph Architecture

The knowledge graph implements a multi-layered architecture that captures both explicit and implicit relationships in the domain (Figure 4). Entity nodes represent variables, functions, and datasets, while edges capture derivation relationships and dependencies. The graph structure incorporates:

1. **Entity Layer:** Represents fundamental elements, including variables, functions, and datasets. Each entity maintains attributes describing its characteristics and requirements.
2. **Relationship Layer:** Captures connections between entities, including derivation paths, dependencies, and implementation patterns.
3. **Validation Layer:** Implements rules and constraints to ensure CDISC compliance and adherence to ADMIRAL best practices.

3.5 Model Training and Optimization

The training process implements a carefully designed approach to fine-tune the LLaMA 3.1 8B model while maintaining efficiency and performance. We utilize Low-Rank Adaptation (LoRA) with the Unsloth PEFT framework to achieve effective domain adaptation while minimizing computational requirements.

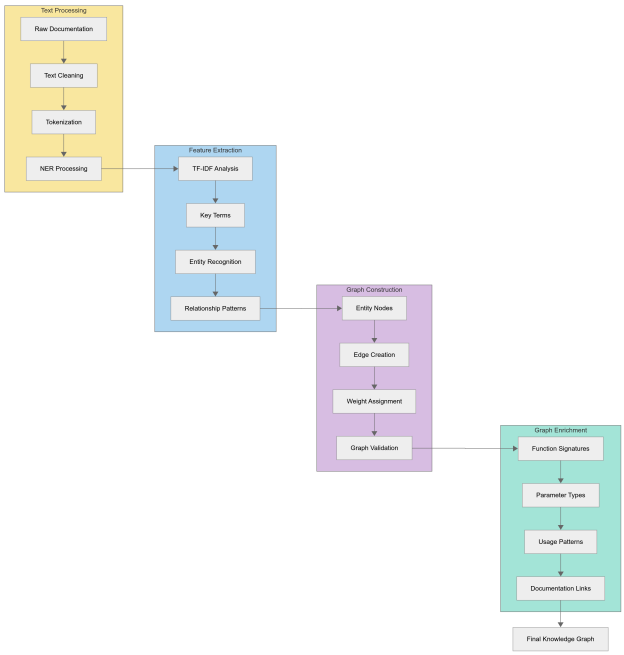

Figure 3: Knowledge Graph building flow

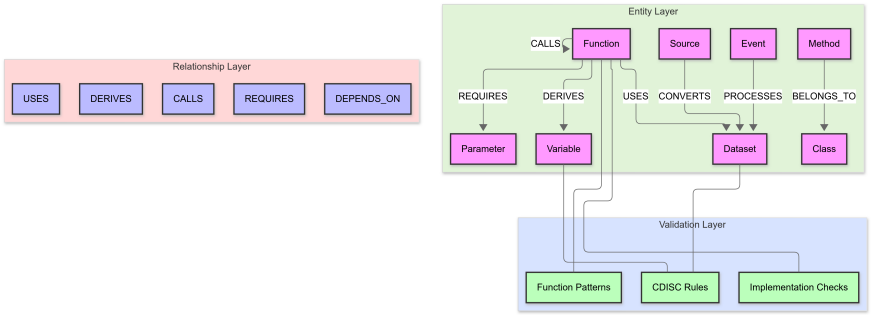

Figure 4: Knowledge Graph Architecture showing entity relationships and validation pathways

3.5.1 Training Configuration

The training configuration optimizes several key parameters:

Model Architecture We utilize the LLaMA 3.1 8B model as our base architecture, implementing LoRA with rank (r) = 16 and alpha () = 32. These parameters were determined through empirical testing to provide optimal performance while maintaining reasonable computational requirements. The model was trained on an L4 GPU with 22.5GB memory and 53.0GB system RAM.

Training Data and Processing The training dataset combines processed ADaM specifications with augmented examples generated through our dual enhancement strategy. Key configurations include:

- Maximum sequence length: Optimized for our question-answer format
- Dataset processing: Implemented with 2 parallel processes for efficient data loading
- No sequence packing to ensure training stability

Optimization Parameters We implement an efficient training strategy with the following configurations:

- Optimizer: AdamW 8-bit with learning rate 2e-4 and weight decay 0.01

- Batch Processing: Per-device batch size of 4 with 4 gradient accumulation steps
- Training Schedule: Linear learning rate scheduler with 5 warmup steps
- Maximum Steps: 1000 training steps
- Mixed Precision: Automatic selection between FP16 and BF16 based on hardware support

Table 1: Training Configuration Parameters

Parameter	Value
Base Model	LLaMA 3.1 8B
LoRA Rank (r)	16
LoRA Alpha (α)	32
Learning Rate	2e-4
Weight Decay	0.01
Batch Size	4
Gradient Accumulation	4 steps
Warmup Steps	5
Maximum Steps	1000
Optimizer	AdamW 8-bit
LR Scheduler	Linear
Random Seed	3407

Hardware Utilization The training infrastructure was optimized for efficient resource utilization:

- GPU: NVIDIA L4 (22.5GB VRAM)
- System RAM: 53.0GB
- Precision: Automatic BF16/FP16 selection based on hardware capability
- Memory Efficiency: 8-bit optimizer for reduced memory footprint

This configuration achieves a balance between training efficiency and model performance, enabling effective fine-tuning while maintaining reasonable computational requirements. The careful selection of hyperparameters and optimization strategies ensures stable training behavior and consistent model performance across different ADaM specification complexities.

4 Performance Evaluation Framework

The proposed evaluation framework encompasses a sophisticated dual-stage methodology designed to ensure both the quality of training data and the comprehensive assessment of the fine-tuned model. At its core, the framework leverages an advanced Knowledge Graph Query System integrated with a multi-metric evaluation strategy, enabling robust model performance measurement.

4.1 Knowledge Graph Query System Architecture

Central to our methodology is the Knowledge Graph Query System (Figure 5), which implements a novel dual-validation mechanism. This system synthesizes structured knowledge graph queries with Large Language Model (LLM) based validation to systematically assess the correctness and reliability of generated Admiral code. The dual-validation approach provides complementary perspectives on code quality, significantly enhancing the robustness of our evaluation process.

4.1.1 Validation Components

Our validation framework comprises two primary scoring mechanisms, each designed to capture distinct aspects of code quality:

1. **Knowledge Graph Score (S_1):** This component evaluates the semantic and structural alignment between generated output and existing knowledge structures through:

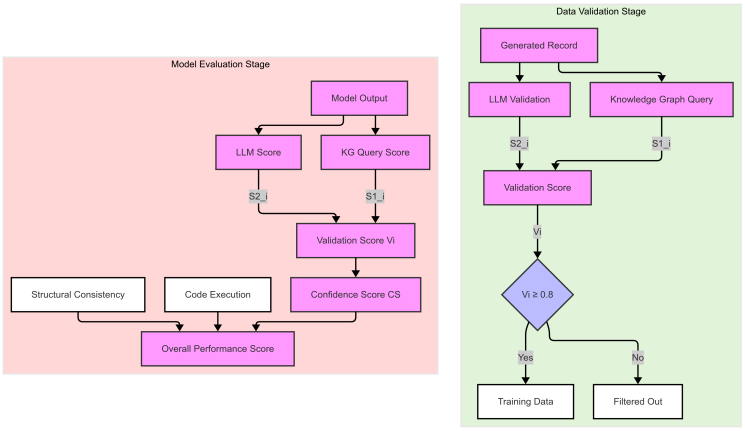

Figure 5: Knowledge Graph Query System structure

- **Vector Similarity Analysis:** A sophisticated measure ensuring semantic proximity to established derivation patterns, utilizing a theoretically-grounded threshold ($\theta > 0.3$)
- **Deterministic Term Matching:** Implementation of precise matching algorithms against validated dataset specifications, yielding binary outcomes
- **Relationship Path Analysis:** Rigorous verification of derivation logic against established knowledge graph dependencies
- **Graph Traversal Evaluation:** Systematic assessment of structural coherence through alignment with validated transformation patterns

2. **LLM Validation Score (S_2):** This component implements a comprehensive assessment of correctness through advanced language model reasoning, incorporating:
   - **Function Usage Analysis:** Systematic evaluation of Admiral function implementation correctness
   - **Parameter Verification:** Rigorous validation of function argument compatibility and correctness
   - **Regulatory Compliance Assessment:** Detailed evaluation of adherence to CDISC standards
   - **Implementation Pattern Coherence:** Quantitative analysis of alignment with established best practices

The final validation score for each generated instance i is computed through a weighted combination:

$$V_i = w_1 S_1^i + w_2 S_2^i \tag{1}$$

where w_1 and w_2 represent empirically optimized weights that balance the contributions of graph-based and LLM-based validation components.

4.2 Multi-Stage Validation Process

4.2.1 Training Data Validation

The training data preparation phase implements a stringent validation protocol:

- Validation threshold: Records must satisfy $V_i \geq 0.8$
- Strict filtering: Non-conforming records are systematically excluded
- Quality assurance: Process ensures optimal training data fidelity

4.2.2 Comprehensive Model Evaluation

The post-training evaluation framework incorporates three complementary metrics:

1. **Confidence Score (CS):** A robust measure of validation consistency across the output space:

$$CS = \frac{1}{N} \sum_{i=1}^{N} V_i \tag{2}$$

2. **Code Execution Accuracy (CA):** A deterministic measure of functional correctness:
$$CA = \frac{P}{T} \tag{3}$$
where P denotes successfully executed instances and T represents the total generation set.
3. **Structural Consistency (SC):** A sophisticated measure of implementation alignment:
$$SC = \frac{1}{N} \sum_{i=1}^{N} Sim(G_i, R_i) \tag{4}$$
where $Sim(G_i, R_i)$ represents the cosine similarity between generated code G_i and reference implementation R_i.

4.3 Integrated Performance Assessment

The Overall Performance Score (OPS) synthesizes the evaluation components through a weighted formulation:

$$OPS = \alpha CS + \beta CA + \gamma SC \tag{5}$$

The weighting coefficients, determined through extensive empirical validation, are:

- $\alpha = 0.4$ (validation confidence weighting)
- $\beta = 0.35$ (execution success weighting)
- $\gamma = 0.25$ (structural alignment weighting)

5 Experimental Results

5.1 Evaluation Setup

We evaluated our approach using ADaM datasets from a single Phase II cardiovascular study. A total of 75 variables were selected across ADSL, ADAE, ADLB, ADTTE, ADVS, and ADRS datasets. The complexity levels of these variables were assessed by experienced ADaM programmers based on the Admiral derivation steps required, with considerations for:

- Number of source datasets required
- Complexity of derivation logic
- Number of Admiral functions needed
- Dependencies between variables
- Required data transformations

The variables were classified into three complexity levels:

5.1.1 Basic Complexity (30 variables)

Variables requiring straightforward Admiral derivations, typically involving:

- Single source dataset transformations
- Basic variable derivations using standard Admiral functions
- Minimal data manipulation steps

5.1.2 Intermediate Complexity (25 variables)

Variables requiring moderate Admiral processing, including:

- Multi-step derivations using Admiral functions
- Time-related calculations and windowing
- Parameter-value combinations

5.1.3 Complex Complexity (20 variables)

Variables requiring sophisticated Admiral implementations, such as:

- Multiple source dataset merging and processing
- Complex derivation logic with multiple conditions
- Chained Admiral function calls
- Advanced data windowing and period calculations

5.2 Performance Metrics

Table 2 presents the comparative performance across different models using the weighted OPS formula ($\alpha = 0.4$, $\beta = 0.35$, $\gamma = 0.25$):

Table 2: Model Performance Comparison

Metric	Fine-tuned LLaMA	GPT-4o	Base LLaMA
Confidence Score (CS)	0.82	0.91	0.42
Code Execution Accuracy (CA)	0.85	0.94	0.35
Structural Consistency (SC)	0.79	0.88	0.28
Overall Performance Score (OPS)	0.82	0.91	0.36

5.3 Analysis by Dataset Complexity

Table 3 shows the OPS breakdown by derivation complexity:

Table 3: Performance by Complexity Level

Complexity Level	Fine-tuned LLaMA	GPT-4o	Base LLaMA
Basic (30)	0.85	0.93	0.45
Intermediate (25)	0.81	0.90	0.32
Complex (20)	0.76	0.87	0.25

6 Discussion

The experimental results demonstrate both the potential and limitations of applying fine-tuned language models to ADaM dataset creation. Several key implications emerge from our findings:

6.1 Model Architecture Implications

The substantial performance gap between GPT-4o and our fine-tuned LLaMA model (OPS difference of 0.09) suggests that model scale and pre-training breadth remain crucial factors in handling complex programming tasks. However, the significant improvement of our fine-tuned model over the base LLaMA (ΔOPS = 0.46) validates the effectiveness of domain-specific adaptation, particularly for specialized frameworks like Admiral.

6.2 Complexity-Performance Relationship

The observed performance degradation across complexity levels reveals an important pattern: while all models show reduced effectiveness with increasing complexity, the rate of degradation varies significantly (GPT-4o: 6.5%, Fine-tuned LLaMA: 10.6%, Base LLaMA: 44.4%). This pattern suggests that:

- Model robustness to complexity correlates strongly with model scale and training breadth
- Domain-specific fine-tuning provides substantial resilience against complexity-induced performance degradation
- The relationship between task complexity and model performance appears non-linear, with accelerating degradation at higher complexity levels

6.3 Error Pattern Analysis

The distribution of errors across different categories (parameter specification: 2.1%, multiple dataset handling: 4.3%, complex logic: 8.2%) highlights specific areas requiring attention:

- Parameter specification errors suggest the need for enhanced validation mechanisms
- Multiple dataset handling challenges indicate potential limitations in context window utilization
- The concentration of errors in complex logic implementations points to opportunities for specialized training approaches

7 Future Work

Based on our findings, several promising directions for future research emerge:

7.1 Model Architecture Enhancement

- Investigation of hybrid architectures combining transformer-based models with symbolic reasoning components
- Development of specialized attention mechanisms for handling complex Admiral function dependencies
- Exploration of multi-task learning approaches incorporating CDISC standard compliance as auxiliary objectives

7.2 Training Data Optimization

- Creation of synthetic ADaM specifications targeting identified error patterns
- Development of automated data augmentation techniques for Admiral code generation
- Integration of real-world ADaM programming patterns through active learning approaches

7.3 Validation Framework Enhancement

- Implementation of automated test case generation for complex Admiral derivations
- Development of specialized metrics for assessing CDISC standard compliance
- Integration of formal verification techniques for critical derivation logic

8 Conclusion

This research presents a comprehensive methodology for automating ADaM dataset creation through the fine-tuning of large language models, specifically targeting the Admiral package ecosystem. Our approach demonstrates that while state-of-the-art models like GPT-4o achieve superior performance (OPS: 0.91), carefully fine-tuned smaller models can provide a viable alternative (OPS: 0.82) for organizations with data privacy constraints or limited computational resources.

The substantial performance improvement achieved through domain-specific fine-tuning (ΔOPS: 0.46) validates our hypothesis that targeted adaptation can significantly enhance model capabilities in specialized domains. However, the observed performance degradation across complexity levels (ranging from 6.5% to 44.4%) underscores the continuing challenges in handling intricate ADaM specifications.

Our knowledge graph-based validation framework provides a robust mechanism for ensuring the reliability of generated code, while the identified error patterns offer valuable insights for future improvements. The methodology established in this work not only advances the field of automated statistical programming but also provides a foundation for future research in domain-specific language model applications.

The implications of this work extend beyond ADaM dataset creation, suggesting broader applications in automated programming for regulated industries. Future work should focus on enhancing model architectures, optimizing training data generation, and strengthening validation frameworks to further improve the reliability and efficiency of automated code generation systems.

Our findings contribute to the growing body of research on domain-specific applications of language models, while also highlighting the practical considerations and tradeoffs involved in their deployment. This work establishes a framework for future investigations into automated statistical programming, particularly in contexts where regulatory compliance and data privacy considerations are paramount.

References

[1] et al. Xu. Automatic instruction evolving for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6999–7012, 2024.

[2] Michael Han Daniel Han and Unsloth team. Unsloth, 2023.

[3] 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.

[4] Yujia Li, David Choi, Brahmni Seneviratne, and Andy Zeng et al. Hubert Tsai. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[5] Edward J Hu, Yelong Shen, Phillip Wallis, and Allen-Zhu et al. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[6] Daniel Martin et al. Katz. Gpt-law: Exploring legal document analysis using gpt models. *arXiv preprint arXiv:2305.12345*, 2023.

[7] Jinhyuk et al. Lee. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 2020.

[8] J. Yan, C. Su, and C. Shi. Ai-enhanced chatbot for streamlined clinical trials analysis and document management. In *PhUSE US Connect 2024*, IC08, 2024.

[9] Yu et al. Wang et al. KEPLER: A unified model for knowledge embedding and pre-trained language representation. In *Proceedings of ACL-IJCNLP*, 2021.

[10] Miltiadis et al. Allamanis et al. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81–92, 2018.

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