Paper ML15

Coding Clinical R with Confidence: Digital AI Assistant based on Self-Hosted LLMs

Kostiantyn Drach, University of Barcelona/Intego Group, Barcelona, Spain
Iryna Kotenko, Intego Group, Maitland, USA

ABSTRACT

Programming in R is gaining momentum in the clinical domain (e.g., Pharmaverse initiative). The rise of AI-powered tools is accelerating this trend, yet there are currently no well-known pre-trained LLMs specifically designed for this domain. To close this gap, we develop a self-hosted digital AI assistant built on coding LLMs (CodeLlama, StarCoder2, etc.), with expert-driven customization and evaluation. We leverage our over-decade experience in training clinical programmers globally to transform it into AI training workflow, targeting industry standards and predictable quality of code produced. Self-hosted models ensure privacy and full control across all stages, making them suitable for regulated environments. The framework combines retrieval-augmented generation (RAG), supervised fine-tuning (SFT), and continuous expert oversight to achieve quality of outcomes, controlled, verified, and benchmarked through multiple layers (human experts, specialized LLMs, etc.). This paper demonstrates how self-hosted, expert-supervised LLMs can deliver reliable R programming support, aligning AI efficiency with quality and compliance.

1. INTRODUCTION

Programming in R is becoming increasingly central in clinical development due to the growth of validated open-source ecosystems and industry initiatives promoting standardized workflows. In parallel, large language models (LLMs) and artificial intelligence (AI) coding assistants are transforming software development. However, generic AI tools are often not suitable for regulated clinical environments due to limited domain specialization, lack of transparency in training data, and data governance constraints.

The goal of this work is to present a proof of concept for a clinical R coding assistant built on self-hosted coding LLMs. Using open-weight models deployed in controlled infrastructure enables full control over data, training, and evaluation, making the approach compatible with regulatory and privacy requirements. In this paper, we outline the architecture, data strategy, and evaluation approach for such a system, and demonstrate its feasibility through targeted experiments.

The system is grounded in trusted knowledge sources, including open-source clinical R package documentation and open-source technical books. These resources form the basis of the RAG pipeline, ensuring alignment with industry standards and reproducible programming practices.

One of the key innovations of this paper is the integration of an internal R-AI-Challenge, run within the organization, designed to capture high-quality code from experienced R and SAS clinical programmers. This initiative produced a curated database of expert-validated solutions that can be used for both RAG and SFT. This represents a shift "*from training people to training models,*" enabling scalable knowledge transfer.

This framework is designed to be transferable to other organizations by converting internal training expertise and validated codebases into structured AI training resources for domain-specific assistants.

2. LLMs for digital AI assistant

2.1. Closed-weight models vs open-weight models

Closed-weight LLMs, developed by companies such as OpenAI®, Anthropic®, and Google®, currently represent the forefront of AI technology. Models such as GPT, Claude, and Gemini are generally delivered via application programming interface (API) access and are operated under strict vendor control, without public access to model weights.

The main advantage of the closed-weight models is their strong performance. Trained using very large datasets and supported by substantial computing infrastructure, they frequently achieve leading results across benchmarks and real-world applications. In addition, vendors invest heavily in safety engineering to reduce harmful outputs and improve behavior on sensitive topics.

At the same time, API-based access requires users to send data to external infrastructure for processing. The internal mechanisms of these models are not accessible ("black box"), which limits transparency, auditability, and customization. Users also have limited influence over model updates, which are fully controlled by the provider. For these reasons, the closed-weight models are used in this work only as comparative baselines.

Open-weight LLMs provide an alternative paradigm. In this work, we use the term "open-weight" to refer to models whose weights are publicly available and can be deployed for local or *self-hosted inference* (e.g., via Ollama), while their training data

and full training pipelines are typically not disclosed. In the literature, the term "*open-source LLM*" is sometimes used interchangeably with "open-weight LLM," even though this usage does not strictly align with traditional open-source definitions, where training code and data are also expected to be available.

The open-weight LLMs vary considerably in their intended scope and specialization. Some models are designed as general-purpose language models, while others are explicitly optimized for specific tasks, such as code generation and program analysis. Code-oriented models are typically trained on large corpora of source code and technical documentation and are better suited for structured programming tasks than general-purpose alternatives.

Representative examples of the open-weight LLMs include Code Llama, Codestral, gpt-oss, etc. These models enable organizations to run inference on private infrastructure, adapt models to specific use cases, and integrate them into environments with strict data privacy requirements. This approach also reduces dependency on external providers and eliminates ongoing API costs.

However, the open-weight models typically require substantial technical expertise to deploy, optimize, and maintain. They may also lag behind leading closed-weight models in raw performance and certain advanced capabilities, which motivates the need for additional techniques, such as retrieval augmentation, to enhance their effectiveness in practical applications [1]**.**

2.2. Enhancing performance of open-weight LLMs

The performance of open-weight LLMs can be improved using two primary approaches:

Fine-tuning [2] involves modifying the model itself by updating its weights through additional training on domain-specific data. In the context of LLMs, fine-tuning is typically used to adapt a general-purpose base model to a specific task or domain using supervised learning on curated input–output pairs. This approach can improve overall model quality, specialize a model for a particular application, or adjust generation behavior and style.

RAG [3] focuses on enriching model prompts with external contextual information in order to improve output accuracy. Given a user query, a RAG system retrieves relevant information from a pre-built knowledge base and injects this context into the prompt provided to the LLM. The model then generates a response based on both its internal knowledge and the retrieved context.

When comparing fine-tuning and RAG, several trade-offs should be considered. Fine-tuning is generally more computationally expensive and requires high-performance hardware and high-quality training datasets, although inference with a fine-tuned model is typically faster. RAG, by contrast, offers greater flexibility, as updating an external knowledge base is usually easier and faster than modifying model parameters, but it requires careful optimization of embedding models, vector storage, and retrieval pipelines.

For the present study, we adopt the RAG approach due to its inherent flexibility for rapid knowledge updates and implementation simplicity. This serves as an initial phase in our research, while the impact of supervised fine-tuning remains a subject for subsequent investigation.

2.3. LLM selection

The evaluation included several open-weight LLMs for code generation:

- gpt-oss-20b (OpenAI),
- Code Llama 13B Instruct (Meta AI),
- StarCoder2 Instruct (BigCode),
- WizardCoder 33B v1.1 (WizardLM Team),
- Codestral 22B (Mistral AI).

We also used selected closed-weight LLMs solely as baselines, namely o3 (OpenAI) and Claude Sonnet 4 (Anthropic). The open-weight models differ in scale and specialization but share the common property that their model weights are publicly available and can be deployed in self-hosted environments. While their licenses vary, they generally permit research use and, in many cases, commercial deployment, albeit without full transparency of training data or pipelines.

To evaluate code generation quality, we used HumanEval-R, an adaptation of the HumanEval benchmark [4] originally introduced by OpenAI to assess a model's ability to generate functionally correct code, rather than merely syntactically plausible solutions. The R-specific version, published on Hugging Face, available at https://huggingface.co/datasets/nuprl/MultiPL-E, is comparable in structure and purpose to the original Python-based benchmark and is designed for R software development.

HumanEval-R consists of 161 tasks, each defined by a natural language problem description, an R function signature, and a set of automated unit tests written in R. Correctness is determined exclusively through execution-based testing.

Model performance is evaluated using the pass@**k** metric, which measures the proportion of tasks for which at least one of **k** generated solutions passes all unit tests.

In this research, only the most conservative variant of the metric, pass@1, was used, where each task is considered solved only if the first generated solution passes all unit tests. The results obtained under this evaluation protocol are summarized in **Table 1**.

Table 1. Comparison of LLMs

LLM	pass@1	runtime
Open-weight models		
gpt-oss:20b	**0.714**	2h 46m
codellama:13b-instruct	0.255	2h 25m
starcoder2:instruct	0.379	1h 27m
wizardcoder:33b-v1.1	0.416	4h 5m
codestral:22b	0.398	2h 16m
Closed-weight models		
o3	0.739	
claude-sonnet-4	0.72	

Overall, **gpt-oss-20b** emerged as the best-performing open-weight model. Although it is not the fastest model in terms of runtime, it achieved the highest pass@1 score among open-weight candidates and demonstrated performance comparable to that of closed-weight models. Model gpt-oss-20b is an open-weight, reasoning-oriented large language model released by OpenAI. It is based on the Transformer architecture and employs a Mixture-of-Experts (MoE) design, in which only a subset of model parameters (experts) is activated for each token, substantially reducing computational cost during inference. The model was trained on a large-scale text-only corpus comprising trillions of tokens, with a strong emphasis on STEM domains (science, technology, engineering, and mathematics), programming, and general knowledge. gpt-oss-20b has a knowledge cutoff of June 2024 [5].

2.4. Selection of Embedding Models

Embedding models are a core component of retrieval-based systems such as RAG. They map textual inputs into fixed-size dense vector representations, enabling semantic similarity search through vector distance measures. In the context of RAG systems, embeddings are used to represent both user queries and documents in a shared vector space, allowing relevant context to be retrieved efficiently based on semantic proximity rather than lexical overlap.

For the retrieval component of the RAG system, several open-weight embedding models were evaluated, including:

- nomic-embed-text (Nomic AI),
- jina-embeddings-v2-base-code (Jina AI),
- granite-embedding (IBM®),
- oscardp96/medcpt-query (biomedical research community),
- mxbai-embed-large (Mixedbread AI),
- embeddinggemma (Google).

To assess embedding quality, a custom retrieval benchmark was constructed. The source corpus consisted of technical documentation from R packages hosted in the pharmaverse repository (https://pharmaverse.org/), which was preprocessed and segmented into chunks. For a randomly selected subset of 200 chunks, one question per chunk was automatically generated using an LLM, ensuring a strict one-to-one correspondence between each query and its relevant documentation fragment. The resulting benchmark comprises of pairs *(query, relevant chunk)* and is designed to evaluate how well embedding models capture semantic similarity between user queries and domain-specific technical text.

For each query, retrieval was performed over a vector database built with a given embedding model, returning the top three most relevant chunks. Retrieval performance was evaluated using Mean Reciprocal Rank (MRR) and Hit Rate. MRR reflects how highly the relevant chunk is ranked on average, favoring models that place the correct document near the top of the result list, while Hit Rate measures the proportion of queries for which the relevant chunk appears among the top three retrieved results. Together, these metrics provide a concise assessment of both ranking quality and practical retrieval effectiveness. The quantitative results of this comparison are summarized in **Table 2**.

Table 2. Comparison of embedding models

Embedding model	Runtime	MRR	Hit Rate
nomic-embed-text	43.33 s	0.9517	0.980
jina-embeddings-v2-base-code	47.92 s	0.9342	0.965
granite-embedding	30.56 s	**0.9700**	**0.990**
oscardp96/medcpt-query	41.34 s	0.5375	0.620
mxbai-embed-large	57.96 s	0.9600	0.980
embeddinggemma	68.16 s	0.9317	0.960

Overall, granite-embedding achieved the best performance among the evaluated models, demonstrating the highest values for both MRR and Hit Rate while also exhibiting the lowest runtime. This indicates a favorable balance between retrieval accuracy and computational efficiency.

2.5. RAG pipeline

Our approach of the RAG flow selected for enhancing performance of open-source models is illustrated in **Figure 1**.

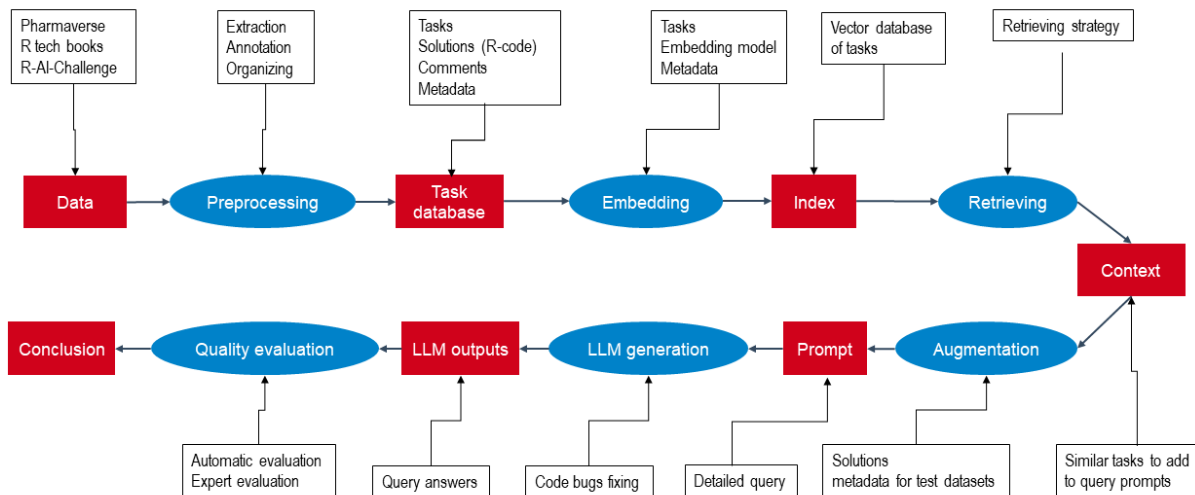

Figure 1. RAG Pipeline

The pipeline starts with collecting **Data**. The high-quality R-code samples are collected from pharmaverse packages used in CDISC-compliant regulatory submissions, published R-tutorials, and technical books on clinical programming[1], and from the special internal **R-AI-Challenge** (see the description below) run within the organization.

The step of data **Preprocessing** includes extracting R-code samples and annotating them by the tasks they solve. The data is organized into a **Task Database**, where each row corresponds to some tasks and columns include task formulation, its solution, comments, and metadata. During the R-AI-Challenge, the tasks were formulated by senior programmers of the company and were solved by the company employees. For the R-tutorial data, first R-code snippets (solutions) were extracted, and the LLM was asked to formulate the task these code snippets solve. For the pharmaverse data, the functions descriptions were the tasks, and the function specification and usage examples were the solutions.

The next step is **Embedding** of the task formulations only to get the vector database (so-called **Index**). Each task is a single chunk. Thus, tasks similar to a user query can be selected, regardless of the code used to solve these tasks. The solutions of the tasks are collected and saved in a separate database. The embedding model also takes into consideration some metadata, including a task type, subtype, and therapeutic area. The embedding model we use in our experiments is the *granite-embedding* model (https://ollama.com/ibm/granite-embedding).

The **Index** is a database of numerical representations of documents (vectors) that stores helpful information for the future LLM generation. To obtain the complete and accurate output from the LLM in response to a specific query (which was certainly not included in the LLM's training dataset), the model should be provided with the context relevant to the query, which helps it to generate the response. We use a Chroma index from the LlamaIndex Python library (https://www.llamaindex.ai/) to make the vector task database.

Retrieving is a process of extracting relevant and adequate **Context** from the Index. We use the hybrid retriever, combining the search results of the Chroma index engine and the BM-25 retriever. The hybrid retriever was selected among other retrievers based on MRR and Hit Rate metrics. The Chroma retriever takes the user query, transforms it into a numerical vector, and searches for the closest vectors in the index. Also, the BM-25 retriever searches for the tasks with the same keywords. The response of the hybrid retriever consists of the top relevant tasks found by both engines.

After retrieving tasks similar to the user query, the solutions of the tasks are taken from the Task Database to provide appropriate code examples to the LLM during the **Augmentation** step. In case the user query is dataset-dependent, the metadata of the dataset, including standard Analysis Data Model (ADaM) or Study Data Tabulation Model (SDTM) variable descriptions, are provided as well. As a result, we incorporate the augmented context into a **Prompt** that includes instructions for the LLM, the task it should perform, and the code samples it should follow.

[1] The sources used are: https://atorus-research.github.io, https://genentech.github.io, https://cli.r-lib.org, https://gsk-biostatistics.github.io, https://rlang.r-lib.org, https://pharmaverse.github.io, https://stringr.tidyverse.org/, https://r4csr.org/

During the **LLM generation** step, we obtain an **LLM output** as an R-code snippet to respond to the user query. Since this code may be incomplete due to the specificity of the dataset (variable names, dataset path, etc.) or erroneous, the step of bug fixing should be done manually or in an automatic way.

The last workflow step is the **Quality evaluation** of the LLM outputs, which is performed in two ways: manually by human experts and automatically using some specially constructed benchmarks.

2.6. R-knowledge databases for RAG

A knowledge database of R code samples was collected from various sources and organized as a dataset in which each row corresponds to an individual code sample. Each code sample was treated as a **task-solution pair**. Each task was characterized by the following components:

- Metadata: a therapeutic area; a task type such as ADaM, SDTM, Tables, Listings, Figures, General; a task subtype (e.g., database specification, summary statistics, treatment groups);
- Task statement;
- Assumed input data;
- Solution: R code;
- Comments: examples and explanations;
- Source: e.g., package documentation, a solution author, and a reviewer.

i) Open-source packages documentation. Documentation from more than 80 open-source R packages recommended by pharmaverse, including admiral, dplyr, tidyr, pharmaverseadam, rtables, and others, was processed for inclusion in the R-knowledge database. Function descriptions were treated as task formulations, while their specifications and usage templates were treated as solutions. Metadata was generated using an LLM. In total, approximately **4,200** tasks were collected.

ii) Open-source R tutorials. Approximately 10 open-source R technical books, study repositories, and framework collections (recommended by pharmaverse and the internal company study program) were processed, including resources such as www.r4csr.org, www.rlang.r-lib.org, www.atorus-research.github.io/pharmaRTF.html. Typical code snippets were extracted as task solutions, and an LLM was used to formulate corresponding task descriptions. In total, approximately **2,700** code snippets were collected.

iii) R-AI-Challenge. The R-AI-Challenge was an internal company event aimed at collecting high-quality R code samples written by company specialists. Tasks for the R-AI-Challenge were formulated by senior statistical programmers based on routine clinical study workflows. The tasks addressed the Clinical Data Interchange Standards Consortium (CDISC) standards (ADaM, SDTM) and TLF generation (tables, listings, and figures). Approximately 100 tasks across three complexity levels were proposed to company R specialists, who were given one week to submit solutions via a dedicated platform. Due to the task selection and submission policy, all tasks were solved. Submitted solutions were reviewed by R experts and assigned quality-level scores. All tasks except one received at least one high-quality solution, and the total number of correct solutions was approximately **250**. Unlike tasks derived from open-source documentation and tutorials, the R-AI-Challenge problems were designed to go beyond simple “one-line” code examples. The resulting Challenge task database is therefore expected to serve as a valuable source of examples for LLM-based code generation in more complex statistical programming scenarios.

3. Experiment

3.1. RAG with open-source packages documentation

To evaluate the performance of the RAG system, a specialized set of tasks was constructed, focusing on the use of functions from R packages hosted in the pharmaverse repository. The primary goal of this evaluation was to assess the system’s ability to correctly apply domain-specific documentation during R code generation.

In most cases, each task corresponded to a single function from a specific R package. Task formulations followed a unified structure and included:

- a textual description of the required operation;
- a description of the input data tables provided as metadata (including column names and their semantics);
- specification of new columns or variables to be created, when applicable.

The system prompt, which contained global instructions applicable to all tasks, explicitly stated that the input datasets were already loaded into the execution environment and should be used directly, without reloading or reinitialization. In addition, a single standardized name for the output object, out_dataset, was enforced to ensure consistency across tasks and to simplify automated evaluation.

For each task, canonical R implementation was provided as part of the benchmark. This reference code generated an output dataset named derived_dataset and served as the ground-truth solution for comparison with the code produced by the LLM within the RAG system. The evaluation procedure was implemented as a unified executable script comprising the following steps:

1. loading the input datasets;
2. executing the canonical code to generate the reference dataset derived_dataset;
3. executing the LLM-generated code to produce the dataset out_dataset;
4. automatically comparing the two datasets using the comparedf function from the arsenal R package.

This approach enabled an objective and reproducible evaluation of the generated code based on output data rather than textual similarity.

Impact of RAG context on code generation. For code generation within the RAG system, the gpt-oss-20b model was used, as it was selected in the previous stage as the most effective open-weight LLM. The retrieval component was implemented as a hybrid retriever, combining dense vector search using the granite-embedding model (via the Chroma index engine) with a BM-25 sparse retriever, enabling the system to leverage both semantic similarity and lexical matching. To assess the contribution of the retrieval component and quantify the effectiveness of the RAG approach, testing was conducted under two conditions:

- without additional retrieved context;
- with additional context automatically retrieved from package documentation by the RAG system.

This comparison made it possible to isolate the impact of the external domain knowledge on code generation quality.

Behavior without retrieved context. The results showed that gpt-oss-20b does not possess intrinsic knowledge of specialized R packages from the pharmaverse repository (such as admiral, sdtmval, sdtm.oak, tidyCDISC, among others). In the absence of additional context, the model typically solved tasks using general-purpose data manipulation packages, primarily from the tidyverse ecosystem (most notably dplyr), rather than the domain-specific functions required by the task. Consequently, the generated code was often syntactically valid and logically plausible but failed to meet domain-specific expectations and clinical programming requirements.

Behavior with retrieved context and domain-specific conventions. Providing additional context in the form of retrieved documentation significantly increased the likelihood that the model would select the correct packages and functions. However, in many cases, this context alone was insufficient to ensure correct usage.

A key challenge was that many clinical R packages (especially those in the admiral and tidyverse ecosystems) use a programming style where variable names are passed as symbols rather than strings, and expressions are constructed programmatically. Without explicit guidance on this style, the model often treated variable names as quoted text or built expressions incorrectly, resulting in errors or semantically incorrect code.

To address this issue, an explicit tidy-evaluation rule block was added to the system prompt. This block clarified how variable names and expressions should be passed (e.g., using symbols rather than strings and wrapping multiple variables or expressions in exprs()), and specified that character strings should be used only for data values, not for column references.

Incorporating these rules substantially improved the correctness of function usage from the pharmaverse ecosystem and increased the overall robustness of the RAG-based code generation pipeline.

To quantify the impact of retrieval augmentation, we evaluated the RAG system under two settings: with retrieved context (RAG) and without retrieved context (no RAG). Performance was measured using the strict pass@1 metric, reflecting single-shot code generation correctness, as shown in **Table 3**.

Table 3. Performance under settings

Setting	pass@1
No RAG	0.2
RAG	0.6

3.2. RAG with R-AI-Challenge data

The following case studies demonstrate the possibilities of RAG enhancement for code generation in some cases.

Case study 1. The next example is the code generation for the following task:

Produce summary tables of adverse event frequencies by System Organ Class (SOC) and Preferred Term (PT). The summaries should support application of standard analysis filters, including but not limited to:

- *Treatment-Emergent Adverse Events (TEAEs)*
- *Serious Adverse Events (SAEs)*
- *Adverse Events related to study treatment*

Other analysis subsets as required (e.g., by outcome). Assume that the ADAE dataset contains at a minimum the variables of AESOC (System Organ Class) and AEDECOD (Preferred Term). Additional ADAE variables available for filtering include TRTEMFL, AESER, AEREL, AEOUT, and related flags.

This task is a modified task from the R-AI-Challenge, and its reference output includes the summary level and the frequencies of adverse events by the AESOC and AEDECOD variables.

The RAG-free gpt-oss produces the erroneous code with problems in accessing nonexistent objects. After fixing bugs by the LLM, the execution of the code still demonstrates the misunderstanding of the problem: the output consists of adverse event frequencies by the AESOC variable and treatment group, but not by AEDECOD (Preferred Term).

The RAG-generation by gpt-oss produces the correct output by the required variables (AESOC, AEDECOD), but the path to the test dataset should be specified manually.

Case study 2. RAG evaluation for a complex case

The following complex, multi-step clinical programming task was designed to evaluate RAG in a realistic setting.

Identify baseline records in ADLB using the last observation before dosing (Baseline flag ABLFL = 'Y'). Derive post-baseline change variables CHG (AVAL – BASE) and percent change PCHG for all lab parameters. Summarize the mean and standard deviation of AVAL and CHG at each visit for each treatment arm. Output a table suitable for clinical reporting.

This task was applied to the CDISC Pilot 01 dataset (ADLBC) under three conditions: (i) no RAG, (ii) RAG with R-AI-Challenge contextual data, and (iii) RAG with R-AI-Challenge contextual data and dataset metadata.

Shared Structural Issue. All three solutions initially encountered BASE column duplication during joins (BASE.x / BASE.y conflicts), requiring manual removal of the pre-existing variable before joining. This issue is orthogonal to RAG and reflects a common data wrangling pitfall when joining datasets with overlapping column names.

No-RAG Solution: Systematic Domain Knowledge Gaps. The no-RAG solution exhibited multiple critical failures. First, this solution demonstrates tooling mismatches, namely attempting to read SAS XPT transport files using read_csv() instead of haven::read_xpt(), and computing change-from-baseline manually (mutate(CHG = AVAL - BASE)) rather than using validated pharmaverse functions (admiral::derive_var_chg()). While the manual formula was mathematically correct, this approach lacks regulatory validation documentation and standardized edge-case handling essential for submissions. Second, this solution shows limited CDISC awareness (e.g., attempting to use non-existent or inappropriate variables like STUDYDAY instead of ADT) and assuming the treatment variable (TRT01P) existed in ADLB without explicitly joining it from ADSL. This suggests that, in the absence of contextual grounding, the model defaults to generic data assumptions rather than domain-specific conventions. Finally, the solution included baseline rows in change-from-baseline summaries – 59 parameter-treatment combinations where CHG = 0 by definition, resulting in extra output rows. As it is not a cosmetic issue, it will be most likely flagged in a regulatory review.

RAG with R-AI-Challenge Data: Substantial Improvement. RAG with domain context substantially improved performance. The solution correctly performed the following operations: (1) used haven::read_xpt() with convert_blanks_to_na(), (2) employed CDISC-standard variables (ADT, AVISIT, PARAMCD), (3) applied admiral derivation functions with regulatory traceability, and (4) implemented post-baseline filtering via conditional baseline assignment (BASE = if_else(ADT >= TRTSDT, BASE, NA)) followed by filter(!is.na(CHG)), producing the correct output. For correct execution, the produced code required a TRT01A treatment variable, which was conceptually correct but indicated incomplete ADLB dataset structure knowledge, because TRT01A is an ADSL variable and not an ADLB one. After this fix (and after resolving the shared structural issue with BASE column duplication mentioned above), the output tables were submission-ready. Verification confirmed numerical correctness: 0 rows with mean CHG differences > 0.01 when compared to the metadata-enhanced version.

RAG with R-AI-Challenge Data and Metadata: Best Structural Reasoning. The metadata-enhanced configuration achieved strongest performance in structural reasoning. Critically, it correctly inferred that treatment information (TRTP) was present in ADLB, eliminating the ADSL join entirely and demonstrating accurate understanding of ADaM dataset dependencies. It also employed the most explicit post-baseline filter (filter(ADT >= TRTSDT)), making analytical intent immediately transparent.

The solution encountered a janitor::clean_names() issue that converted CDISC-standard uppercase variables to lowercase, requiring the transformation to be disabled. After this fix, the output was numerically identical to the RAG-only version (0 CHG differences > 0.01) with minor n-count differences (1–3 patients per group, 194 total rows) stemming from different handling of patients with missing actual treatment labels. Both approaches are valid depending on analysis population specifications.

Implications. This case study demonstrates that RAG is essential for domain-specific clinical programming. Without RAG, the model produced superficially plausible but analytically unsafe code that violated fundamental reporting conventions. RAG enabled correct tooling selection, CDISC compliance (standard variable codes, post-baseline logic), and structural reasoning (dataset dependencies), submission readiness via using admiral-derived statistics.

CONCLUSION

In this work, we presented a practical framework for building domain-specific coding assistants in regulated environments using self-hosted open-weight LLMs.

We selected an optimal open-weight coding model and embedding model specifically for R programming, based on execution-based code correctness and retrieval quality metrics. Based on this choice, we developed a RAG pipeline tailored to clinical R programming, where correctness is defined not only by functional output but also by adherence to domain standards (e.g., CDISC conventions, validated package usage). The experiments show that retrieval context is necessary for domain-correct code generation and substantially improves pass@1 performance compared to standalone model inference.

The knowledge base combines approximately 7,000 open-source clinical R snippets together with approximately 250 expert-validated solutions collected through the R-AI Challenge run within our company. This provides a scalable mechanism to continuously inject high-quality company-grade code into the system, that can be adapted to other organizations.

A key advantage of combining RAG with the R-AI Challenge data is that it injects expert-validated, workflow-realistic clinical programming knowledge directly into the generation process. Unlike open-source documentation or tutorial snippets, the R-AI Challenge tasks are designed by senior programmers around real submission workflows (e.g., CDISC-compliant derivations, TLF generation, multi-step dataset transformations). As a result, retrieved context from this source helps the model move

beyond syntactically correct code toward **regulatory-safe and submission-ready implementations**, including correct tool selection (e.g., validated pharmaverse functions), correct dataset logic, etc.

Our work naturally raises a key data strategy question: how to optimally combine large volumes of lower-quality, elementary data with smaller volumes of high-quality, domain-specific, complex code. Solving this balance is critical for building scalable but reliable domain AI assistants.

REFERENCES

[1] K. Drach and I. Kotenko, "Scaling up: accelerating self-hosted open-access LLMs to compete with the Big Three (ChatGPT, Claude, Gemini)," in *Proceedings of PHUSE US Connect 2025 (March 16-19, 2025)*, Orlando, USA, 2025.

[2] Y. Yu, S. Zuo, H. Jiang and oth., "Fine-Tuning Pre-trained Language Model with Weak Supervision: A Contrastive-Regularized Self-Training Approach," *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* pp. 1063-1077, 2021.

[3] P. Lewis, E. Perez, A. Piktus and oth., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *NeurIPS Proceedings,* vol. 33, pp. 9459-9474, 2020.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan and oth., "Evaluating Large Language Models Trained on Code," 2021.

[5] S. Agarwal, L. Ahmad, J. Ai, S. Altman and oth., "gpt-oss-120b & gpt-oss-20b Model Card," 2025.

ACKNOWLEDGEMENTS

We would like to acknowledge **Oleksandr Leonov** (Kharkiv National University, Ukraine), **Lyudmyla Polyakova** (Kharkiv National University, Ukraine), and **Viktoriia Shevtsova** (Intego Group, Ukraine) for being core members of the research team and handling the computational experiment. We would also like to acknowledge **Viktor Hladun** and **Oleksandr Andreichikov** (Intego Group Data Science Lab), who helped in gathering data and preparing the R-AI-Challenge platform as well as all the participants of the R-AI-Challenge. Without you, this research and the experiment would not have been possible.

AI tools (ChatGPT 5.2, 2025; Claude 4.5 Sonnet, 2025) were used for editing and spell-checking of this paper.

CONTACT INFORMATION

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

Contact: Sergey Glushakov
Company: Intego Group
Address: 2300 Maitland Center Pkwy, Suite 240, Maitland, FL 32751, USA
Work Phone: +1 407 512-1006
Email: sergey.glushakov@intego-group.com
Web: www.intego-group.com

Brand and product names are trademarks of their respective companies