LLMs for Preprocessing Neural Time-Series Data

Sanvi Pal
California Institute of Technology
Pasadena, CA
spal2@caltech.edu

Alexa Baxter
California Institute of Technology
Pasadena, CA
abaxter@caltech.edu

Amrita Pasupathy
California Institute of Technology
Pasadena, CA
apasupat@caltech.edu

Maya Keys
California Institute of Technology
Pasadena, CA
mekeys@caltech.edu

Abstract

While large language models (LLMs) are becoming increasingly capable of generating code, their effectiveness remains limited in domains with ever-evolving technical standards. One such domain is EEG preprocessing, which is a foundational step in BCI research. It requires domain-specific expertise and lacks a single standardized protocol, making it difficult to generalize across datasets or rely on code templates. In this work, we apply retrieval-augmented generation (RAG) to EEG preprocessing tasks, enabling the integration of up-to-date documentation and field-specific practices during code generation. Building on the CODERAG-BENCH framework, we curate a benchmark tailored to EEG workflows that draws from MNE documentation, GitHub repositories, and Stack Overflow discussions for domain-specific context. Our system supports both canonical and open-retrieval settings and evaluates robustness, syntax validity, and task achievement of our outputs using OpenAI's GPT-4o model. We find that structured prompts and question-specific retrieval significantly enhance the reliability of the produced code, although challenges remain in handling ambiguous queries and generating fully functional code. We hope our project helps facilitate further research in the use of RAG for scientific and biomedical domains.

1 Introduction

RAG or retrieval-augmented generation is a method that was introduced by Facebook AI (1). The RAG pipeline has two components: the retriever and the generator. The retriever is an example of non-parametric memory where an external knowledge source that is queried at runtime to retrieve relevant information. In RAG, common retrievers are DPR (Dense Passage Retriever) and vector databases like FAISS, which search a large corpus (like Wikipedia, documentation, GitHub code) to return top-k relevant passages based on a query. The generator is an example of parametric memory which refers to the knowledge stored in the weights of a neural network learned during pre-training and fine-tuning. Examples of generators are Bigstarcoder, Mistral, and OpenAI's GPT-4o. These models have been trained on a large text corpus and are capable of generating fluent outputs based on what they've implicitly learned and stored in their parameters during training. In other words, their knowledge is embedded in the neural weights of the model.

The "augmentation" part of RAG comes from the fact that the generator is not generating text based solely on its internal (parametric) knowledge. Instead, it conditions its output on retrieved external documents, effectively combining what it knows with what it retrieves. This hybrid architecture allows RAG systems to generate responses that are both fluent (thanks to the parametric model)

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

and factual or context-aware (thanks to the non-parametric memory). For example, in tools like LangChain, this architecture is operationalized by embedding the user's query, retrieving relevant context chunks from a document store (like FAISS or Pinecone), and passing both the query and context into a model like GPT-4 to generate an informed response.

RAG has several advantages: it enables models to adapt to new information without retraining, provides transparency and traceability for outputs (since retrieved sources are available), and significantly reduces hallucinations in knowledge-intensive domains.

Many works in academia have applied RAG to text corpora and currently, companies are applying RAG to their internal documents. One application of RAG that has been underexplored is code-generation (2). Many code tasks, especially in domains like EEG preprocessing, require domain-specific knowledge. RAG allows LLMs to expand their knowledge with up-to-date implementations in a specific domain, which helps mitigate hallucinations.

We model our pipeline closely to a recent NLP publication, "CODERAG-BENCH: Can Retrieval Augment Code Generation?". Instead of their corpus, we applied RAG to four relevant EEG repositories. We parsed all the repositories to BEIR format, ran canonical and open retrieval, and ran generation with OpenAI GPT-4o.

We chose the following repositories as our corpus:

- The Python `mne` documentation
- The ICLR 2025 paper *"CBraMod: A Criss-Cross Brain Foundation Model for EEG Decoding"*
- `PyPREP`, a Python implementation of the Preprocessing Pipeline (PREP) for EEG data
- `EEG-pyline`, an EEG preprocessing pipeline in Python

2 Links to code & demo

Demo Notebooks

- **Canonical retrieval notebook:** `https://colab.research.google.com/drive/1mrqxg_F7dJNWkNi088yIs3XUYL8661Xv?usp=sharing`
- **Open retrieval notebook:** `https://colab.research.google.com/drive/1atFGomyJCbrI2S-9OK4QvcANrtvGu9nM?usp=sharing`
- **EEG data experiment notebook:** `https://colab.research.google.com/drive/1JXk2TGh3RdKOLO2GBv8yX0XwuqZUNRbu?usp=sharing`

Github Repository

- **eeg_preprop_rag:**
  `https://github.com/sanvi1855544/eeg_preprop_rag`

3 Background and previous work

3.1 Background on EEG Analysis

Time-series data analysis is relevant across many different fields: financial forecasting, weather forecasting, and decoding neural signals. Almost every raw time-series dataset has artifacts that can impede good analysis. Thus, before any forecasting or classification is applied to time series data, reseachers must do signficiant data preprocessing. The problem is that preprocessing procedures are not only inconsistent across different fields but are inconsistent across the same modality. We propose a project where the LLM proposes preprocessing code via a one-shot/few shot code generation. We are not expecting this tool to fully standardize the preprocessing procedure for time-series data, but are proposing an approach that can 1) help efficiently implement preprocessing and 2) offer suggestions for improvement.

We are creating this tool specifically for Electroencephalography (EEG) data, to help Brain Computer Interface (BCI) research. BCIs are systems that interpret or influence brain activity by directly interacting with its electrophysiological signals. Specifically, "read-out" BCIs record neural activity, using either invasive or non-invasive methods. Invasive approaches involve surgically implanted microelectrode arrays that measure neural firing directly. Non-invasive methods, such as surface electrode recordings, detect signals from the scalp or nearby skin. EEG is a noninvasive modality that records electrical activity on the brain from electrodes placed on the scalp.

One major application of EEG recordings is decoding neural signals associated with movement. Machine learning models trained to decode these signals in real-time can lead to robust prosthetic devices that enable users to control their movement through their brain activity. Thus, this field offers promising advances for individuals with motor disabilities, providing a direct neural interface to assistive tools. This project is inspired with the intent to contribute to the BCI field and the potential advances the field is working towards.

EEG data has many motion artifacts, muscle activity, and environmental interference (3). Therefore, the process of cleaning the data is quite time-consuming and relies on signal processing expertise and judgment. If the preprocessing step is done incorrectly, it can remove or obscure meaningful signals, leading to poor model performance for downstream tasks. Thus, while it is true that for every unique dataset, preprocessing can't be fully standardized, our tool aims to make the process more efficient and ensures that the steps that are widely considered standardized in the BCI field included. Most of the innovation machine learning BCI researchers are interested in is in the decoding section of the classification pipeline. Thus, having this tool streamline the time spent cleaning datasets can help many machine learning researchers focus on decoding contributions.

For example, models like TOTEM (TOkenized Time Series EMbeddings for General Time Series Analysis) process time series data to generate embeddings, which can be used for tasks like classification (4). However, this model that scales well across many different kinds of time series data requires well-preprocessed time series data as input. Thus, a model that automates and validates preprocessing can be paired with existing models such as TOTEM to enable efficient end-to-end time series analysis.

EEG data has many motion artifacts, muscle activity, and environmental interference (3). Therefore, the process of cleaning the data is quite time-consuming and relies heavily on the programmer's expertise and judgment. If the preprocessing step is done incorrectly, it can remove or obscure meaningful signals, leading to poor model performance for downstream tasks. Thus, while it is true that for every unique dataset preprocessing can't be fully standardized, our tool aims to make the process more efficient and ensures that the steps that are widely considered standardized in the BCI field are included. Most of the innovation machine learning BCI researchers are interested in is in the decoding section of the classification pipeline. Thus, having this tool streamline the time spent cleaning datasets can help many machine learning researchers focus on decoding contributions.

3.2 Previous Work

Several efforts have been made to simplify and standardize EEG preprocessing, and yet the problem remains mainly unsolved, especially for researchers working with diverse datasets.

One attempt at establishing a preprocessing standard is PyPREP, a Python implementation of the PREP pipeline, developed in MATLAB by Bigdely-Shamlo et al. (5). This codebase aims to standardize EEG preprocessing by automating a sequence of robust, commonly accepted preprocessing steps. Specifically, the codebase outlines functions for channel rejection and filtering techniques.

While this was an admirable effort, the problem is that PyPREP or PREP wasn't widely accepted by the BCI field as a standard because there is no way to set a rigid standard for preprocessing when datasets themselves have so much variability (caused by the subjects, resampling, or setup of the recording paradigm itself). What can be done though, is the introduction of a tool that greatly streamlines preprocessing workflows and 'learns' from the approaches of existing papers/codebases.

EEG data is prominently processed via the MNE-Python library. Along with referring to mne documentation to learn how to use its built in methods for filtering, researchers can adopt open-source pipelines from GitHub repositories (fine-tuning parameters like filter settings, rejection criteria, and epoching schemes). However, this process requires significant domain expertise to ensure that the

preprocessing is both accurate and effective (6). It also is time consuming because it requires 1) keeping up with new effective procedures and 2) cross-referencing many codebases to extrapolate the most relevant preprocessing approaches.

The rise of large language models has brought new possibilities for assisting with code generation. These models can reproduce common code patterns, allowing them to quickly generate large amounts of working code. However, they are usually only evaluated on general-purpose programming tasks rather than domain-specific ones. For example, CodeGen, a well-known code generation model, excels at performing basic coding tasks but lacks the high-level understanding of neural data and its related preprocessing techniques that would be necessary for EEG preprocessing or BCI applications (7).

A promising step toward domain-specific application is EEG-GPT, a model that demonstrated the ability to classify EEG recordings as normal or abnormal based on verbalized summaries of extracted features such as band power and kurtosis. While EEG-GPT focused primarily on classifying "bad/noisy" data, it illustrates how language models can be effectively applied within the EEG domain. Building on this foundation, our approach extends the role of language models beyond classification by leveraging them for code generation and preprocessing procedure validation. This application is particularly important for preprocessing tasks where context (documentation or procedures already accepted in the field) matters (8).

We are using the context-awareness from Retriever-augmented generation (RAG) to enable efficient preprocessing code for BCI pipelines. This will enhance the robustness and efficiency of BCI preprocessing workflows.

4 Research question and approach

Our research investigates how LLMs can be leveraged to improve the preprocessing stage of brain-computer interface (BCI) pipelines (specifically for EEG data) through context-aware code generation. Preprocessing is a critical but time-intensive step in EEG analysis, involving tasks such as filtering, event marking, and artifact rejection. We investigate whether Retrieval-Augmented Generation (RAG) systems can help automate and refine these tasks by producing contextually informed code snippets to be used in neural data analysis.

Our project poses two central research questions:

1. How can LLMs be used to make preprocessing for brain-computer interface (BCI) pipelines more efficient?
2. How can we use Retrieval-Augmented Generation (RAG) for context-based code generation?

To address these questions, we extend the CodeRAG-Bench framework — an open-source benchmark for evaluating RAG-based code generation — by customizing it for EEG preprocessing tasks. CodeRAG-Bench originally supports a variety of general software engineering tasks using a RAG pipeline that combines retrieval from provided documentation and forums with transformer-based generation. We build on this infrastructure to support querying and code generation for domain-specific challenges of EEG preprocessing.

4.1 Domain-Specific Dataset Curation

One of our first challenges was constructing a dataset suitable for EEG tasks. Unlike traditional coding domains, where high-quality question-answer pairs are abundant (especially on quorum websites such as Stack Overflow), EEG preprocessing resources are scattered across different sources of documentation, tutorials, and GitHub repositories. To address this, we curated a domain-specific dataset by combining code and documentation from the PyPREP repository (a Python implementation of the PREP pipeline), CBraMod, mne documentation, and EEG-pyline. These sources provide real-world examples of EEG preprocessing steps, including bad channel detection, signal filtering, and resampling.

To align this dataset with the CodeRAG-Bench framework, we started by parsing it into the BEIR standard format. Each entry is our `corpus.jsonl` file contained a code segment and description for a single preprocessing method, while queries in `queries.jsonl` captured user-style questions

about how to perform certain operations (for example, one included question is: "How do I apply ICA to remove eye blinks?"). Relevance labels between the corpus and queries were stored in `qrels/test.tsv`. This format allowed us to plug into the benchmark's retrieval pipeline and conduct systematic evaluations. The CODERAG-BENCH paper had their datasets annotated with questions/prompts so we annotated our queries to have natural language questions instead of documentation.

4.2 Retrieval Infrastructure and Embedding Choice

For the retrieval step, we evaluated two approaches, *canonical* and *open* retrieval, to compare their effectiveness. Canonical retrieval operates on a fixed set of documents that were predetermined to be relevant for the current problem, promoting precision while limiting the scope of available data. In contrast, open retrieval allows searching over a broader corpus, which offers greater flexibility at the cost of introducing irrelevant or noisy documents. We specifically expanded our corpus by combining our four repository corpus with RepoEval, which was the better performing repository in CODERAG-BENCH (2). We explored both approaches to understand the trade-offs between narrow expertise and generalization in the context of EEG preprocessing code retrieval.

In both canonical and open retrieval, we implemented a dense embedding approach using the pretrained Sentence Transformer model `sentence-transformers/all-MiniLM-L6-v2` and stored the resulting document embeddings. We then evalated the retrieval performance by utilizing the existing scripts `eval_beir_sbert_canonical.py` and `eval_beir_sbert_open.py` to compute cosine similarity scores between the embedded documents and their queries.

In the beginning, we experimented with the CodeBERT-base model due to its training on both natural language and code, which led us to expect better performance. However, the model yielded suboptimal retrieval results, which may be attributed to the fact that it is not specifically designed to generate high-quality sentence embeddings. Ultimately, the MiniLM model produced a superior embedding space, likely due to its optimization for creating high-quality sentence embeddings.

A key implementation detail in our retrieval system is that we embed queries at runtime rather than using precomputed embeddings, which allows for greater flexibility: users can input natural-language queries outside of our training sets and still receive accurate results. This approach reflects real-world usage, where EEG researchers might have different ways of phrasing questions or might ask about newly developed preprocessing strategies.

4.3 Code Generation with Contextual Prompts

To support EEG preprocessing code generation within the CodeRAG-Bench framework, we developed a custom task class, PyprepTask, which extends the base Task class provided by the evaluation infrastructure. This class handles the construction of context-enriched prompts to be passed into our base model, GPT-4o, for code generation by embedding the original query and finding the top-k semantically similar code segments in the corpus. These retrieved segments are concatenated to form a rich context window, which is then fed into the language model.

We experimented with several decoder models for generation and ultimately chose GPT-4o due to its strong performance on long-context prompts, consistent syntax generation, and ability to follow structured instructions. We initially tested a smaller model: HuggingFace's StarCoder 1B because it is less expensive but it performed poorly. We then attempted StarCoder2 7B, which had a limited context window that prevented us from fully evaluating its performance. Given these constraints, 4o became an appealing choice due to its hosted API availability and ability to handle arbitrarily long or complex prompts.

An important finding from our iterative process was that the quality of the prompts and the relevance of the retrieved context were just as important as the capabilities of the LLM itself. Early versions of our system suffered from mismatched prompts or irrelevant retrievals, leading to hallucinations or incorrect code suggestions.

4.4 Challenges and Design Reflections

Notably, we perform query embedding dynamically within the model rather than relying on precomputed top-k corpus embeddings stored for each query, similar to the other Task classes. This design

choice increases the flexibility of our model, allowing the user to input arbitrary queries beyond those present in the original dataset. This enhances real-world applicability by allowing users to leverage our EEG RAG pipeline for their own custom queries.

5 Results and Experiments

5.1 Retrieval Results

Source	NDCG@1	NDCG@5	MRR@1	Recall@5	P@1
CBraMod	0.0175	0.0822	0.0351	0.1404	0.0175
mne	0.7733	0.8761	0.7733	0.9550	0.7733
PyPREP	0.7750	0.8962	0.7750	0.9750	0.7750
Combined	0.7431	0.8414	0.7445	0.9174	0.7431

Table 1: Evaluation of canonical retrieval performance by dataset

We evaluate the canonical retrieval performance of our system in various standard metrics. These metrics measure the ranking quality (NDCG), the accuracy of top-ranked results (MRR), the ability to retrieve relevant documents among the top results (Recall), and the proportion of relevant documents in the top-ranked positions (Precision).

Our results are seen in Table 1. Notably, the CBraMod dataset performed particularly poorly. This is likely attributed to the fact that its corpus solely contained code without any documentation, making it more difficult for the embedding model to match the queries with the relevant code context.

The combined dataset merges all the above sources together. While it has slightly lower metric values than some individual models, it yields overall strong results. The decline in scores is likely due to the increased size and added noise from variations in the query and corpus formatting and specificity. However, these factors also contribute to improved generalizability, which is crucial for accommodating a wider range of questions.

5.2 Prompt Management

To guide our Large Language Model towards generating high-quality EEG preprocessing code, we developed and evaluated three prompt engineering strategies.

5.2.1 Experiment 1: Direct Instruction Prompting

The first approach utilizes a structured, instruction-based prompt to generate a complete, runnable preprocessing script. Our model, GPT 4o, is provided a clear role to play: an expert Python developer working with MNE-Python to simulate and preprocess EEG data. This helps the model follow a clear identity and task-oriented framing, ensuring it uses relevant context in producing outputs. The model is explicitly asked to:

1. Simulate 10 seconds of synthetic EEG data with five channels sampled at 1000 Hz using `mne.create_info` and `mne.io.RawArray`
2. Apply a preprocessing pipeline that includes a band-pass filter (1-40 Hz) using the firwin design, average referencing, and Independent Component Analysis with 5 components, and excluding the first 2 components
3. Avoid saving files, using disallowed functions such as `linear_regression_raw`, and plotting or visualizing data
4. Output the code enclodes in triple backticks without any extra explanations

The model is provided with reference documentation snippets retrieved from the vector store and a user input question, and is asked to output working preprocessing code to answer the user's question.

The prompt also includes a template example to help guide the expected output format, equipped with import statements and commented code.

We engineered the prompt to pass each set of requirements, constraints, and examples as distinct blocks with explicit section headers for better parsing and consistency in generated code. We also explicitly asked for a code-only output, to make it easier to extract and evaluate the results.

5.2.2 Experiment 2: Prompt Completion without One-Shot

The second approach introduces a more flexible, context-aware prompting strategy. Rather than inputting a question and generating a new code file, we input partially completed or broken code, and ask the model to output the complete, running version of that code.

We provide the model with the same reference documentation from Experiment 1. The model is also given a slightly different set of preprocessing constraints, where the model is asked to avoid deep learning code and classifiers, to avoid any visualizations or saving code, and to only use valid MNE preprocessing functions.

This setup allows the model to focus on repairing or completing code in a context-sensitive method, which improves its ability to generalize to a variety of input scripts. However, without a provided example, the model has less of a backbone for what the expected version should look like, and occasionally does not output the correct format or complete code.

5.2.3 Experiment 3: Prompt Completion with One-Shot

To evaluate the effect that provided examples have on output success, this experiment mirrors Experiment 2 but includes a single one-shot example that demonstrates how a broken script should be transformed into a working preprocessing pipeline.

This example serves as a guide to the model's generation, encouraging consistent and well-formatted solutions. We found that the one-shot example improves the model's reliability, especially in incorporating domain-specific functions and functional pipelines.

However, one limitation is a potential overfitting to the structure and contents of the example, especially when the user's input code differs significantly. This may lead to outputs that mirror the example and do not respond to the unique needs of the input question.

5.3 ChatGPT-3.5 as an Evaluator

Due to the quality of our generated scripts, we asked ChatGPT-3.5 to qualitatively rank them in multiple different categories. The first category is robustness, which checks the code's executability when processing expected inputs as well as some unexpected edge cases. The second one is syntax validity, which ensures that all lines are to be valid Python. The third is how well it achieves the task, which is a measure of semantic correctness. ChatGPT outputs these results on a scale from 0 to 10, as well as a final percentage indicating the overall performance based on all three categories.

We then applied this evaluator to Experiment 1, yielding the following results for canonical versus open retrieval.

Category	Score
Robustness	9
Syntax Validity	10
Achieving Task Requirements	9
Overall Percentage: 93%	

Table 2: Canonical Retrieval with Example

Category	Score
Robustness	8
Syntax Validity	10
Achieving Task Requirements	9
Overall Percentage: 90%	

Table 3: Canonical Retrieval without Example

Category	Score
Robustness	8
Syntax Validity	9
Following Requirements & Task	10
Overall Percentage: 90%	

Table 4: Open Retrieval with Example

Categories	Eval Score
Robustness	9
Syntax Validity	10
Task Requirements	10
Overall Percentage: 96.67%	

Table 5: Open Retrieval without Example

5.4 Application of Model-Based Preprocessing on EEG data

In order to test how relevant the generated code can be in helping a BCI researcher improve their preprocessing pipeline. We conducted a test to visualize whether the first 30 channel signals were correctly filtered. We just applied this test on our canonical pipeline.

Due to time constraints, we did not fine-tune the generator component. Instead, we used GPT-4o as a post-processing step to refine the initial generator output based on the retrieved context. We fed the user input describing a minimally processed signal along with the generator output to GPT-4o to get a post-processed script. We then applied that script to a 3 second segment from an EEG recording (Prof. Yisong and Geeling's decoding project, D10S2).

This implementation of this can be seen in our demo (canonical) notebook: https://colab.research.google.com/drive/1JXk2TGh3RdKOL02GBv8yX0XwuqZUNRbu?usp=sharing

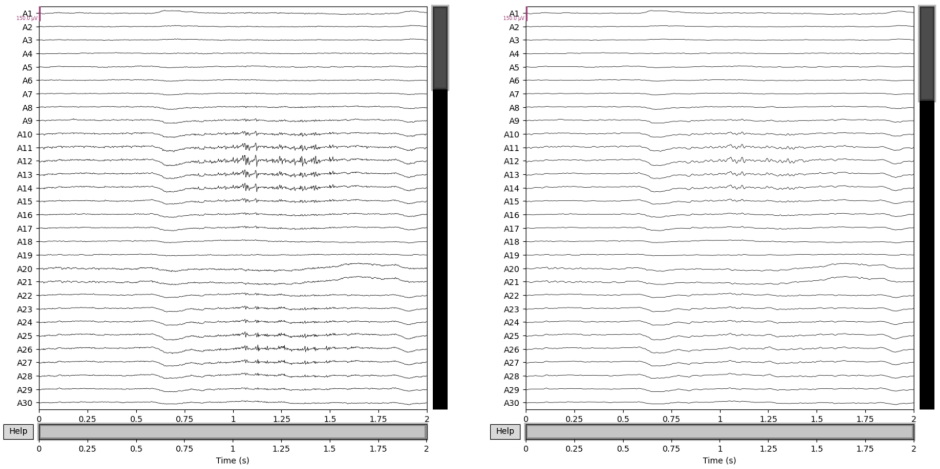

Figure 1: Left: EEG signal from 3s recording, bandpass signal[1, 256], notch filter Right: GPT-4o finetuned script that took in generator input

As seen in the above figure, there is good potential for one-shot preprocessing with RAG if there were to be more finetuning.

6 Discussion

Our results show that retrieval-augmented generation (RAG) is a method that often fails to work as intended. We hypothesized that having a domain-specific corpus would lead to tailored preprocessing but instead dealt with the pitfalls of RAG: the necessity of prompt engineering and finetuning the generator.

The effectiveness of RAG is highly dependent on prompt quality and alignment. Without fine-tuning, even powerful LLMs like GPT-4o struggle to seamlessly incorporate retrieved snippets into accurate code unless the prompt clearly instructs them how to do so. Early versions of our prompts often led to hallucinated code or outputs that ignored the retrieved context entirely. Our output was only one line of code until we significantly reworded the prompt. We noticed how prompt-engineering is tricky because you need to be specific enough that the generator understand how to use the context to execute a task but it cannot be too specific that there is overfitting (we noticed the same output was generated in every run with our third prompting experiment in our demo).

In Experiment 1, we found that providing an example in the prompt affected the evaluation scores of the generated code. From the results in Tables 2 through 6, we noticed that examples helped for canonical retrieval but had the opposite effect on open retrieval. When looking closer, we found that many open retrieval generations that used examples in the prompt would return code nearly identical to that example. This indicates that when examples are paired with open retrieval, the model has a

tendency to overfit, as it can access a more highly customizable corpus that allows to reconstruct the example almost exactly.

Also, with regards to our generation, in our second prompting experiment we expected the output to be our code but with improved preprocessing. Instead, the generator gave ideas and functions that we can add to improve our preprocessing. If we had more time, we would have finetuned the generator or possibly done more experiments with the prompts. However to test the validity of the output of the generator, we inputted the generator output along with information regarding a 3s data segment to GPT-4o to get a working preprocessing script. We ensured that the GPT-4o model was used the ideas outputted by the generator. We demonstrated this script correctly filtered the data as the signals on the right of figure 1 are smoother and don't have artifacts like the channel signals on the left.

Lastly, it is important to note that we initially approached this project with the goal of contributing to the CodeRAG-Bench corpus. However, during our development process, we observed that the majority of datasets included in CodeRAG-Bench were designed around general-purpose code generation problems (toy-level programming tasks or common issues found on platforms like StackOverflow). Furthermore, these datasets were specifically curated and annotated for RAG. They included well-structured question–answer pairs along with corresponding test suites for automatic evaluation.

In contrast, our corpus, drawn from real-world neural data preprocessing code (EEG pipelines), does not come with such annotations or tests. Each function would have required a custom test implementation to support evaluation metrics like pass@k, which are central to CodeRAG-Bench. Due to time constraints, we were unable to complete this extensive annotation and test generation step, which limited our ability to integrate our dataset into the benchmark in a rigorous manner.

7 Conclusion

In this work, we explored the use of RAG for EEG preprocessing, a domain where standards evolve rapidly. Despite its perceived promising framework, we found that RAG struggles without extensive prompt design, high quality retrievals, and finetuned generators.

Given our observed limitations, we want to explore an alternative paradigm: tool-augmented generation. Rather than relying on retrieved context to guide the model, we want to use language models that directly interface with APIs and validated preprocessing tools. This approach would allow the model to reason over structured tool outputs and directly call trusted preprocessing pipelines without needing to reproduce them from memory, which can lead to hallucinations and missed context.

Ultimately, while RAG shows room for improvement, we believe that success in this field requires hybrid systems that blend language models with concrete procedures. For EEG preprocessing in particular, the tool-use approach offers most potential for keeping pace with the field's rapid evolution.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *arXiv preprint arXiv:2005.11401*, 2020.

[2] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried, "Coderag-bench: Can retrieval augment code generation?," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2024.

[3] T. R. Mullen, C. A. E. Kothe, Y. M. Chi, A. Ojeda, T. Kerth, S. Makeig, T.-P. Jung, and G. Cauwenberghs, "Real-time neuroimaging and cognitive monitoring using wearable dry eeg," *IEEE Transactions on Biomedical Engineering*, vol. 62, pp. 2553–2567, Nov. 2015.

[4] S. Talukder, Y. Yue, and G. Gkioxari, "Totem: Tokenized time series embeddings for general time series analysis," *arXiv preprint arXiv:2402.16412*, 2024. Accepted to TMLR (12/24), last revised Jan 2025.

[5] N. Bigdely-Shamlo, T. Mullen, C. Kothe, K.-M. Su, and K. A. Robbins, "The prep pipeline: standardized preprocessing for large-scale eeg analysis," *Frontiers in Neuroinformatics*, vol. 9, p. 16, 2015.

[6] A. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, and C. M. Michel, "Mne software for processing meg and eeg data," *NeuroImage*, vol. 86, pp. 446–460, 2013.

[7] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint*, 2022.

[8] J. Kim, D. Park, and M. Cho, "Eeg-gpt: A deep learning model for classifying eeg recordings based on extracted feature summaries," *IEEE Transactions on Biomedical Engineering*, vol. 71, no. 2, pp. 567–574, 2024.