How does AI code with repository? An evaluation of ICL strategies for repository code generation

Anonymous ACL submission

Abstract

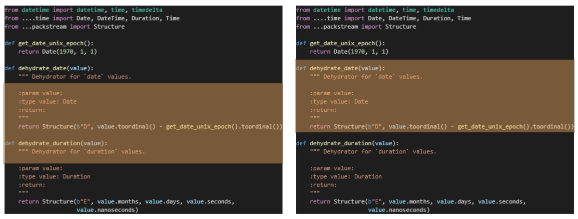
Retrieval-Augmented Generation (RAG) is the main method to embed context information Language Model (LM) pipelines. However, repository-aware code generation pose a challenge to off-the-shelf RAG due to lack of specificity of traditional embedders, usually trained to handle context-inespecific coding benchmarks such as HumanEval, MBPP and APPS. In order to create reliable pipelines, without relying on any retriever or generator finetuning, we studied the impact of different contexts: 1) We firstly include the local scope of the retrieved functions and methods; 2) we extend it to include the whole function file in the context; 3) we evaluate the impact of the implementation in the same file of the new function ("Infile" context); 4) we combine the entire retrieved function file with Infile; finally (5) we evaluate the ability of Language Models (LMs) to self-generate documentation and use them to implement new repository functions. Our experiments show the necessity of keeping the whole current, and retrieved file in the context as opposed to specific methods and classes. With this setup, we reach the state-of-the-art performance in CoderEval benchmark employing the open-source small-scale Llama3.1-8B-Instruct without finetuning the generator or the retriever, and without relying on compiler feedback.

1 Introduction

Large Language Models (LLMs) have extensively been used for code generation (Shinn et al., 2024; Jiang et al., 2023; Chen et al., 2022; Ishibashi and Nishimura, 2024; Bi et al., 2024; Dong et al., 2023; Wang et al., 2023a; Hong et al., 2023). The typical scenario involves the LLM reading a problem statement and implementing a self-contained code or rely on standard or publicly libraries for implementing their solutions (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021). A more challenging scenario rises when we prompt the LLM to generate a code to be inserted in an implemented repository. The LLM must understand the functions, classes, constants, imports, and the general repository architecture to correctly call the necessary functions, classes, and other codes to seamlessly integrate in the existing components. This poses a more challenging scenario than the former one, since the LLM needs to consult and understand a repository, potentially out of its knowledge scope, to generate the correct requested implementation.

While Retrieval-Augmented Generation (RAG) is a common strategy to obtain knowledge or code snippets from the target repository (Lu et al., 2022; Bi et al., 2024; Zhang et al., 2023; Wang et al., 2024), most of this works do not explore different ways of building the context. Traditionally, they employ a slide window-based strategy (Zhang et al., 2023) to scan the lines of the file codes to obtain text *chunks*, which are later embedded into a vector representation using some text processing or encoder (e.g., BM25 (Robertson et al., 1995), UniXCoder (Guo et al., 2022), CodeT5+ (Wang et al., 2023b), etc.). However, Wang et al. (2024) already acknowledge the slide window weakness, and propose a novel *chunks* creation strategy based on the entire function code body; also Feng et al. (2024) already evaluate the positive effect of keeping the current file in the generator context. Pointing to the importance of studying the impact of the context in code generation.

Some state-of-the-art methods improve code generation by training a retriever on generator output (e.g., RLCoder (Wang et al., 2024)) or by using compiler feedback (Bi et al., 2024; Zhang et al., 2023). However, these approaches have significant drawbacks. Training a custom retriever for each repository is costly, unscalable for large projects, and incompatible with proprietary models like GPT-4 (Achiam et al., 2023). This method also creates security risks, as the retriever can embed sensitive information from private code into its param-



```python
from datetime import datetime, time, timedelta
from ....time import Date, DateTime, Duration, Time
from ...packstream import Structure

def get_date_unix_epoch():
    return Date(1970, 1, 1)

def dehydrate_date(value):
    """ Dehydrator for `date` values.

    :param value:
    :type value: Date
    :return:
    """
    return Structure(b"D", value.toordinal() - get_date_unix_epoch().toordinal())

def dehydrate_duration(value):
    """ Dehydrator for `duration` values.

    :param value:
    :type value: Duration
    :return:
    """
    return Structure(b"E", value.months, value.days, value.seconds,
                     value.nanoseconds)
```

```python
from datetime import datetime, time, timedelta
from ....time import Date, DateTime, Duration, Time
from ...packstream import Structure

def get_date_unix_epoch():
    return Date(1970, 1, 1)

def dehydrate_date(value):
    """ Dehydrator for `date` values.

    :param value:
    :type value: Date
    :return:
    """
    return Structure(b"D", value.toordinal() - get_date_unix_epoch().toordinal())

def dehydrate_duration(value):
    """ Dehydrator for `duration` values.

    :param value:
    :type value: Duration
    :return:
    """
    return Structure(b"E", value.months, value.days, value.seconds,
                     value.nanoseconds)
```

Figure 1: *Chunks* comparison. In the left, we have the slide window-based approach, where a slide window of a fixed size w slides every w lines obtaining the text *chunks*. In this case, incomplete parts of different functions can fall in the same window which can mislead the code generation. In the right, as proposed in (Wang et al., 2024), the *chunk* is the entire function body, which avoids incomplete parts and allows the LLM to have an overview of the function.

eters. Similarly, compiler feedback can mislead code generation. It may create overly complex contexts that cause LLM distraction or hallucinations, particularly if the repository's code is highly specialized. Critically, this approach requires running LLM-generated code, which introduces a major security vulnerability by potentially executing malicious code within the project's environment. Thus, in this work, we aim to expand the AI-automated software engineering by answering the following question: *How do we create the LLM context for repository-aware whole-body function generation **without** training the retriever and generator, nor requiring the compiling feedback?*

We believe this research has the following scientific and industrial interests: 1) it will contribute to the on-growing study of In-Context Learning (ICL) for LLMs in a challenging scenario; 2) We do not require severe extra security layers for deployment, nor dedicated hardware for training retriever and generator, and we focus on open-source and small-size language models (e.g., Llama3.1-8B-Instruct (Grattafiori et al., 2024)) saving time and resources; 3) Our solution is plug-and-play and it is easier to be adapted to any public or proprietary retriever or generator. In summary, our main contributions are:

1. We analyze the context impact in repository-aware code generation by considering the main aspects of retrieved functions/methods. We propose and evaluate in the context: 1) functions and methods scope; 2) their whole file; 3) combination with the current file; and 4) self-generated documentation.
2. We propose the RAG of Thoughts (RAGoT), which is an extension of the proposed Chain-of-Thoughts for repository-aware code generation. It achieves a competitive performance to other variants and prior works, and it provides an acceptable trade-off between context size and performance.
3. We **DO NOT** train the retriever and generator, nor rely on compiler feedback, and focus on open-source and small-size language models. Even so, we outperform prior state-of-the-art, which require training of the generator or retriever, in CoderEval (Yu et al., 2024), employing the same generator checkpoints.
4. We show our conclusions stand for different combinations of open-source LLMs and retrievers, easing reproduction and deployment.
5. Our solution is plug-and-play and it is easier to be adapted to any public or proprietary retriever or generator.

2 Related Work

Early research in AI-driven code generation focused on creating standalone functions from natural language descriptions. Some approaches directly prompt LLMs with the task, while others introduce a structured pipeline with planning (Jiang et al., 2023) or agent collaboration (Hong et al., 2023; Ishibashi and Nishimura, 2024). Multi-turn generation, as in CodeGen (Nijkamp et al., 2022), allows the user to decompose tasks into smaller prompts. PanGu-Coder (Christopoulou et al., 2022) uses two-stage training: pretraining on raw text/code and fine-tuning on text-to-code tasks. Role-based systems like Self-Collaboration (Dong et al., 2023) simulate real-world development with agents acting as analyst, coder, and tester. These strategies enhance planning, coordination, and task decomposition. However, they assume the function is generated from scratch without referencing an existing codebase. Realistic scenarios often involve navigating repositories with multiple files and dependencies. In such contexts, understanding and integrating with existing implementations becomes crucial. This shifts the challenge from isolated code synthesis to repository-aware code generation.

Some prior works in code generation have been trained and evaluated in the repository-aware code generation setup. RepoCoder (Zhang et al., 2023) is a framework designed to enhance repository-level code completion by integrating a similarity-based retriever with a pre-trained code language model in an iterative retrieval-generation pipeline. This approach effectively utilizes repository-wide information to generate code at various levels of granularity. RLCoder (Wang et al., 2024) introduces a reinforcement learning framework for repository-level code completion, enabling retrievers to learn without labeled data by evaluating retrieved content based on the perplexity of the target code. It incorporates a stop signal mechanism, allowing the retriever to autonomously decide when to retrieve and which candidates to retain. In ReACC (Lu et al., 2022), the authors propose a retrieval-augmented code completion framework that enhances code prediction by integrating a code retriever with an auto-regressive language model. This approach leverages external code snippets with similar semantics to improve code completion accuracy. In (Bi et al., 2024), the authors introduce CoCoGen to obtain the code execution compiler feedback, which enhances generation through an iterative refinement pipeline. It identifies mismatches between generated code and the project's context using compiler techniques, then aligns and fixes these errors using information from the code repository.

3 Methodology

This methodology will present different ways of approaching code generation. First, we will understand the impact of scope in the LLM context evaluating **RAG-Scope (RAG-S) and FileRAG** approaches. Second we will explain how planing might impact the final generation evaluating the **RAG of Thought (RAGoT)** approach. Third, we will study the impact of the current file in generation (called **Infile**). And finally we will study if using LLM-generated documentation (called **Self-Doc**) will improving context.

RAG, RAG-Scope (RAG-S) and FileRAG. Traditionally, to implement a new function[1] the function signature and its docstring forms the query that is used to retrieve top-k most similar functions in the repository, which are added in LLM context (this procedure is here called as **RAG**, and example of context with $k = 1$ is illustrated in Listing 2.). However, we argue that this RAG strategy is too simple and it does not provide a proper context to the LLM understand the repository to implement the new function. To improve performance, we need to improve the LLM context. Our main goal is to answer the following question: *How do we create the LLM context to obtain a repository-aware knowledge to implement new repository functions, without training the retriever nor the generator?* We hypothesize that the retrieved functions needs its scope. Import statements, variables and constants declarations and other functions and classes implementations in the retrieved functions files may elucidate how a retrieved function from the vector store operates, how its parameters are defined, how it is called, for what it is implemented and how its returning values are used. In summary, functions scope empowers the LLMs context to more effectively generate the new functions. To validate this hypothesis we designed two novel ways to create the LLM context: RAG-Scope (RAG-S) and FileRAG.

When a function f, from a file F, is retrieved from the vector store we have the following setups: **RAG-S** will keep its scope along with its implementation, as well as other functions with the same name of f in F. The rationale is to provide imports statement, variable, constants, block declarations and functions with similar name that surround the retrieved function in its file to better understand its behavior;**FileRAG** will assume the entire file F is important to comprehend f behavior, and insert it entirely in the generator context. RAG-S is a particular case of FileRAG, since RAG-S keeps just part of the file. In counterpart, FileRAG keeps the whole file and consequently the whole local information about where the retrieved function is implemented. We show a context example of RAG-S and FileRAG in Listing 3 and Listing 6, respectively.

RAG of Thought (RAGoT). Previous approach seek to retrieve functions based on the new function signature with its docstring as the query. So, the retrieved functions can be understood as few-shot examples about how to implement the new function. A natural alternative thinking is: the model can try to firstly understand the new implementation by analyzing the function signature and its docstring before implementing it. Similar to the Chain-of-Thoughts (CoT) strategy the generator can break its implementation down in steps, and retrieve the repository functions that mostly closes implement each step. After that, the context will be formed by retrieved functions that implements each intermediate steps of the solution, and later aggregated to prompt the generator. In summary, the generator analyzes the signature and docstring,

[1]The terms 'function' and 'method' will be used interchangeably along this paper

breaks the implementation down in steps, each step is used as query that retrieves the repository functions that mostly close implement it, insert the retrieved function in the context, append the steps used as query, and ask the generator to implement the new function. We call this strategy RAG of Thoughts (**RAGoT**). This is basically an extension of previous RAG, but instead of inserting in the context the retrieved functions based on the function signature and docstring, we insert the functions that implement each step (each "thought") of the model. An example of the generator context for RAGoT can be find in Listing 4. We also employ the same strategy considering each function scope as previously, obtaining RAGoT-S (Listing 5) and FileRAGoT (Listing 7).

Infile. So far we have focused in retrieving functions in the repository to support the new function implementation. So, the retriever retrieves functions considering the whole repository, but may ignore the function file where the new function will be implemented in. Typically, in a repository development scenario we implement new functions on files where it already has some other functions and methods implemented, and that probably have been already tested. In this case, the current file where the new function will be in can provide a local context for the LLM to guide the generation of the new function. We validate this hypothesis by keeping just the current file where the new function will be implemented in as the context, and prompt the generator to implement it. We call this strategy by "Infile". Note that this strategy does not perform any searching, since the context will be the functions already implemented in the current file. A prompt example with this strategy is provided in Listing 8. If the current file is empty, and the new function will be the first one, we fall in the previous cases where all retrieved functions will be searched in the repository.

We can easily fuse the "Infile" strategy by, for instance, keeping the current file and the file retrieved with the "FileRAG" strategy. Then we will have the current file plus another file in the repository, and we call it "Infile + FileRAG". A prompt example with "Infile + RAG" (our proposed "Infile" strategy employed with standard RAG) is in Listing 9.

Self-Doc. Finally, we design a set of experiments to answer the following question: *Can the documentation of functions in the repository help to implement new code?* Typically, when we are working with a new repository or library, we try to look at some documentation, if any, to better understand the repository and its functions, classes, methods, etc. Then we implement new functions and codes based on the documentation. We validate if this strategy is also effective for LLMs. To achieve this, we firstly, retrieve k files from the repository using FileRAG. Then we prompt the LLM to read and understand each function and method in the k files. The LLM outputs each function signature, the respective explanation about what each function does, and what it returns. After that, we insert the generated documentation in the LLM context along with the respective signature and docstring of the function to be implemented. The goal is to check if the own LLM understanding of the functions, compressed in the generated documentation, is enough to support new functions implementation. Also, we will check what happens to the LLM performance when the functions bodies are replaced by their documentation. We call this strategy "Self-Doc". An extension of it is also presented when we ask the LLM to perform RAGoT over the generated documentation, naming it "Self-Doc + RAGoT". In other words, the LLM will break the task in steps based on the self-generated documentation.

4 Experiments

We conduct all experiments using the CoderEval benchmark (Yu et al., 2024), which evaluates code generation in repositories across six sample types, ranging from self-contained to project-level. Our focus is on the more challenging and practically relevant types: class-level, file-level, and project-level samples. CoderEval includes 230 Python samples from 43 repositories and 230 Java samples from 10 repositories. We evaluate performance using the pass@1 metric, which measures the proportion of correct implementations on the first attempt. For code generation, we use the open-source Llama3.1-8B-Instruct model (Grattafiori et al., 2024), and for retrieval, CodeT5+ (110M) (Wang et al., 2023b), due to its strong retrieval performance. We also include comparisons with other small-scale open-source LLMs and retrievers. Our emphasis on open-source, lightweight models reflects practical concerns such as cost-efficiency, privacy, and independence from proprietary APIs. This setup provides a realistic and reproducible evaluation environment for repository-level code generation. We run our experiments in two H100 GPUs with 95GB of memory.

4.1 Ablation study of context strategies

The results for each setup presented in previous section are shown in Table 1. Clearly, RAG states among the worst models with one of the lowest overall average performance (last column). This shows that, differently from other language tasks, RAG is not so effective for repository-aware code generation. Our first strategy to improve the context is RAG-Scope (RAG-S), which shows a performance improvement over RAG in "class" and "file" setup for Java (26.00 vs 18.00 and 100.00 vs 0.00[2], respectively), and in the most challenging project setup (13.04 vs 8.70 for Python and 18.18 vs 11.36 for Java). Overall, RAG-S improves from 11.16 to 31.02 over the standard RAG, with marginal increment in the numbers of context tokens. Our proposed RAGoT slightly underperforms RAG in the "class" Python setting but outperforms it in "file" and "project" setups. It uses an intermediate number of tokens compared to RAG and RAG-S. Overall, RAGoT is less effective than RAG-S. To address this, RAGoT-S incorporates function scope, improving performance in "class" and "project" Java, and showing better overall results than RAG and RAGoT. This improvement comes at the cost of increased context tokens due to additional functions needed for each step.

The natural extension of the scope is to include the whole function file in the LLM context. In this sense, our proposed "FileRAG" achieves the best performance across most evaluation scenarios so far, outperforming other setups by large margins. The FileRAG average Python performance has a superior compared to previous approaches (14.85 of RAGoT vs 26.14 of FileRAG), just loosing for FileRAGoT, but with a smaller margin (26.14 vs 28.41). In Java, the best performance is still attached to RAG-S, since, among all methods in the upper part of table, it is the only one to correctly implement the single "file" Java example. This result pushes the performance up for RAG-S. Also, FileRAG has the highest number of tokens in the context for Python, but keeps competitive with other variants in Java. FileRAGoT extends FileRAG by retrieving an entire file for each reasoning step, rather than just the single most relevant file. This results in multiple files being included in the context, one per step. While FileRAGoT outperforms earlier variants, its performance remains slightly below that of FileRAG.

In the same line, we check the impact of current file (where the new function will be implemented in) in the context, which is the "Infile" setup. From Table 1, it has competitive performance compared to FileRAG and FileRAGoT, showing that, indeed, the local context provides strong support. When we extend Infile with the previous strategies, we see a performance increase for all of them, where "Infile + FileRAG" obtains the best performances. It obtains an overall average python performance of 34.49, where the runner-up is 28.41, and average Java performance of 71.15, where the runner-up is in 60.12. These results show that the local information may be complementary to the other information in the repository.

The last set of experiments evaluate the LLM ability to leverage its own self-created documentation for code generation. From the last five lines of table 1, we see that the self-documentation has limited performance, suggesting that the function body in the context is better than its documentation.

The main conclusions are three fold: 1) The scope plays a fundamental role to provide an understanding to the generator about how the functions work and operate. When the full function scope is included in the context (the full function file) we obtain the best performance, with "Infile + FileRAG" attaining the best results across most of the evaluated setups; 2) The standard RAG (first line) states among the setups with the lowest performance, showing that the **standard RAG is not suitable for repository-aware code generation**; and 3) Break the requirements in steps, similar to CoT, brings improvements in some setups (lines with RAGoT), but it is not as effective as "Infile + FileRAG". In other words, **CoT behaves differently for repository-aware code generation**. Further study are still necessary to successfully integrate CoT in repository-aware code generation.

4.2 Ablation study of text/code embedders

For completeness, in Table 2, we show the results of some of our variants employing other well-know retrievers, such as UniXCoder (Guo et al., 2022), Contriever (Izacard et al., 2021), MPNet (Song et al., 2020) and BM25 (Robertson et al., 1995). We see that CodeT5+, Contriever and MPNet hold similar performances. For Infile + FileRAG, for

[2]In CoderEval, there is just a single example for file_runnable evaluation. So we have a binary performance with 100.00 when the LLM successfully implements it, and 0.00 otherwise

Table 1: Ablation study with Code Generation methods in CoderEval dataset. For each column, the best one is in blue, and the second best is in green. The performance is calculated in terms of pass@1.

			class		file		project		Python Avg.	Java Avg.	Overall Avg.
Method	Avg. # Tokens in Context	Encoder	Python	Java	Python	Java	Python	Java			
			Llama3.1-8B-Instruct								
RAG	0.43K/0.30K	CodeT5+	12.73	18.00	16.18	0.00	8.70	11.36	12.54	9.79	11.16
RAG-S	1.19K/1.04K	CodeT5+	12.73	26.00	16.18	100.00	13.04	18.18	13.98	48.06	31.02
RAGoT	0.89K/0.60K	CodeT5+	10.91	18.00	20.59	0.00	13.04	9.09	14.85	9.03	11.94
RAGoT-S	2.58K/1.91K	CodeT5+	7.27	27.00	17.65	0.00	8.70	15.91	11.21	14.30	12.75
FileRAG	5.94K/1.02K	CodeT5+	34.55	46.00	26.47	0.00	17.39	31.82	26.14	25.94	26.04
FileRAGoT	3.70K/7.37K	CodeT5+	28.57	41.94	40.00	0.00	16.67	23.81	28.41	21.92	25.16
Infile	4.68K/8.22K	-	23.64	44.00	29.41	0.00	17.39	36.36	23.48	26.79	25.13
Infile + RAG	3.91K/7.48K	CodeT5+	20.00	44.00	17.65	100.00	17.39	36.36	18.35	60.12	39.23
Infile + RAG-S	6.06K/9.09K	CodeT5+	23.64	45.00	26.47	100.00	13.04	29.55	21.05	58.18	39.62
Infile + RAGoT	4.37K/7.77K	CodeT5+	27.27	36.00	32.35	100.00	13.04	27.27	24.22	54.42	39.32
Infile + RAGoT-S	6.06K/9.09K	CodeT5+	23.64	39.00	23.53	100.00	17.39	27.27	21.52	55.42	38.47
Infile + FileRAG	9.42K/17.42K	CodeT5+	50.91	68.00	26.47	100.00	26.09	45.45	34.49	71.15	52.82
Self-Doc (current file)	1.01K/1.02K	-	12.73	15.00	13.24	0.00	8.70	6.82	11.56	7.27	9.42
Self-Doc (current file + 1 extra file)	1.98K/1.95K	CodeT5+	14.55	14.00	11.76	0.00	8.70	11.36	11.67	8.45	10.06
Self-Doc (current file + 2 extra files)	2.92K/2.67K	CodeT5+	18.18	18.00	14.71	0.00	4.35	2.27	12.41	6.76	9.59
Self-Doc (current file + 3 extra files)	3.86K/3.30K	CodeT5+	10.91	16.00	10.29	0.00	4.35	4.55	8.52	6.85	7.68
self-Doc + RAGoT-S	2.68K/1.88K	CodeT5+	9.09	27.00	19.12	0.00	8.70	11.36	12.30	12.79	12.55

instance, CodeT5+, Contriever and MPNet has an average Python performance of 34.49, 34.26 and 31.60, respectively. For Java, they achieved pass@1 of 71.15, 65.97 and 65.88, respectively.

For UniXCoder, the pipeline runs out of memory for most of the Java experiments (denoted by "OUT" word in Table 2), with exception to "Infile + RAG-S" which has the lowest number of context tokens, on average, in the context. This shows that our model may be sensible to some embedders and how much context they retrieve. However, it successfully runs for all Python experiments for the four setups. For "Infile-FileRAG", UniXCoder average Python performance is 28.91, the lowest compared to other embedders. Conversely, it has the best performance for "Infile + RAG-S", achieving 23.81 compared to 22.01 from MPNet. Interestingly, similar behavior happens for "Infile + RAG-S" in Java, achieving 58.61 compared to 58.18 from CodeT5+ and MPNet. This shows that some variants can improve performance if a proper retriever is employed.

Finally, BM25 does not show competitive performance, exceptionally for "self-Doc + RAGoT-S" that achieves the best average Python performance, however, it has a substantial lower result in Java. Overall, BM25 does not provide competitive results mainly because it is a shallow retriever, which does not provide enough semantics for code retrieval. In conclusion, we recommend the employment of one of the four neural retrievers, mainly CodeT5+ and Contriever that state among the best performers.

4.3 Comparison with the State of the Art

In this section we compare our two best context formation strategies employed with varied language models. Differently from prior works, we **do not** finetune the retriever and the generator, nor employ compiler feedback or perform multiple rounds of RAG. We evaluate our solutions with four different language models, namely "Llama3.2" (Grattafiori et al., 2024) (1B and 3B versions), "Llama3.1" (Grattafiori et al., 2024) (8B version) that has been employed in all experiments so far along this paper, and DeepSeekCoder-V1 (Guo et al., 2024) (7B version). All are instruct-based models. As our competitors, we employ RepoCoder (Zhang et al., 2023) and RLCoder (Wang et al., 2024), which are among the state-of-the-art models in CoderEval benchmark, and have publicly available codes that allow us to rerun their solutions with the aforementioned language models with the exact the same checkpoints we employed. The results are presented in Table 3.

For Llama3.1-8B-Instruct, our solution "Infile + FileRAG" achieves the state of the art, outperforming RLCoder, with the same language model checkpoint, by 10.91 percentage points (p.p.) and 38.00 p.p. in class level for Python and Java, respectively, by 7.35 p.p. in file level for Python, and by 4.35 p.p. and 20.45 p.p. in project level for Python and Java, respectively. Additionally, we achieve the best Python, Java and overall performance. This shows the positive impact of our proposed "Infile + FileRAG". By properly creating the context, we do not require retriever and generator training, even compared to a model that trains the retriever (RLCoder) and one that employs multiple RAG iterations (RepoCoder).

On average, our best solution requires 9.42K and 17.42K tokens for Python and Java samples, respectively, in CoderEval. For this reason, we also compare our proposed "FileRAG" in Table 3, since it has competitive performance and an av-

Table 2: Ablation study with Code Generation methods in CoderEval dataset for different retrievers (encoders). The performance is calculated in terms of pass@1.

		class		file		project		average		
Method	Avg. # Tokens in Context	Python	Java	Python	Java	Python	Java	Python	Java	Overall
						CodeT5+				
FileRAG	5.94K/10.24K	34.55	46.00	26.47	0.00	17.39	31.82	26.14	25.94	26.04
Infile + RAG-S	6.06K/9.09K	23.64	45.00	26.47	100.00	13.04	29.55	21.05	58.18	39.62
Infile + FileRAG	9.42K/17.42K	50.91	68.00	26.47	100.00	26.09	45.45	34.49	71.15	52.82
self-Doc + RAGoT-S	2.68K/1.88K	9.09	27.00	19.12	0.00	8.70	11.36	12.30	12.79	12.55
						UniXCoder				
FileRAG	6.47K/9.97K	18.18	OUT	29.41	OUT	21.74	OUT	23.11	-	-
Infile + RAG-S	5.26K/8.17K	29.09	44.00	20.59	100.00	21.74	31.82	23.81	58.61	41.21
Infile + FileRAG	9.95K/17.15K	32.73	OUT	27.94	OUT	26.09	OUT	28.92	-	-
self-Doc + RAGoT-S	2.23K/-	10.91	OUT	20.59	OUT	8.70	OUT	13.40	-	-
						Contriever				
FileRAG	5.64K/7.74K	34.55	42.00	25.00	0.00	17.39	25.00	25.65	22.33	23.99
Infile + RAG-S	4.74K/8.14K	29.09	43.00	20.59	0.00	13.04	22.73	20.91	21.91	21.41
Infile + FileRAG	9.12K/14.92K	47.27	57.00	29.41	100.00	26.09	40.91	34.26	65.97	50.11
self-Doc + RAGoT-S	2.62K/2.02K	7.27	19.00	19.12	0.00	8.70	9.09	11.70	9.36	10.53
						MPNet				
FileRAG	5.85K/7.93K	25.45	43.00	17.65	0.00	17.39	25.00	20.16	22.67	21.42
Infile + RAG-S	4.62K/8.09K	23.64	45.00	25.00	100.00	17.39	29.55	22.01	58.18	40.10
Infile + FileRAG	9.33K/15.10K	43.64	59.00	29.41	100.00	21.74	38.64	31.60	65.88	48.74
self-Doc + RAGoT-S	2.43K/1.70K	10.91	23.00	19.12	100.00	4.35	11.36	11.46	44.79	28.12
						BM25				
FileRAG	6.43K/5.34K	38.18	17.00	29.41	0.00	17.39	15.91	28.33	10.97	19.65
Infile + RAG-S	5.14K/7.95K	23.64	46.00	23.53	0.00	17.39	38.64	21.52	28.21	24.87
Infile + FileRAG	9.91K/12.52K	41.82	50.00	26.47	0.00	17.39	29.55	28.56	26.52	27.54
self-Doc + RAGoT-S	3.62K/1.47K	20.00	10.00	17.65	0.00	4.35	2.27	14.00	4.09	9.05

erage token size of 5.94K and 1.02K for Python and Java, respectively (Table 1). Clearly, due to the substantial context reduction, the performance drops compared to "Infile + FileRAG". However, it still attains the second best average Java performance and the second-best overall performance with Llama3.1-8B-Instruct, also outperforming the state of the art. Our strategies are not benefited just from the current file ("Infile" setup) but also due to our proposed RAG strategy to retrieve and keep function file in the context.

Even with smaller models, such as Llama3.2-3B-Instruct, we outperform prior works. With "Infile + FileRAG" we achieve an average Python performance of 22.96 against 22.22 from RepoCoder, and an average Java performance of 23.00 against 14.73 from RLCoder. Consequently, this setup also keeps the best overall performance (22.98). "FileRAG" is the second best. These results are particularly interesting, since when we reduce models size we have more space for context tokens, allowing a more effective usage of our strategies.

However, when reducing too much the model size our strategies have limited performance as evidenced with Llama3.2-1B-Instruct. Our solutions require some minimal level of knowledge from the base language models, a phenomenon that has been already verified by prior RAG literature (Melz, 2023). In this work, the author states that some RAG solutions are only effective in bigger models, since they require a minimal knowledge level typically present on them.

Our solutions show limited performance with models like DeepSeekCoder-7B-Instruct, which struggle when the context exceeds 4096 tokens, leading to illogical outputs. This negatively impacts results. In contrast, LLaMA models handle overloaded contexts more robustly, maintaining reasonable performance despite larger input sizes.

For completeness, our final analysis compare our two best strategies with chatGPT (GPT-3.5), and other repository-aware code generation models employing it as language model. The results are depicted in Table 4. Even with a much smaller language model our solutions has competitive performance compared to GPT-3.5-based solutions. One can state that our model is better due to data leakage, however, we argue that, if so, it has limited impact. We argue based on: 1) All solutions under the same language model name in Table 1 operates with **exact the same** language model checkpoint. It means that RLCoder, RepoCoder and our setups under the Llama3.1-8B-Instruct name, operates with exact the same language model checkpoint. Even so, our solutions outstand compared to others; 2) The "Direct" lines in Table 1 indicates that the model was directly prompt without any context. If any sample from CoderEval was memorized, it would be able to still generate successfully the correct answers, which does not happen to any language model. In fact, GPT-3.5 has better "Direct" performance than Llama3.1-8B-Instruct on

Table 3: Comparison with relevant repository-aware Code Generation methods in CoderEval dataset. For each language model, the best one is in blue, the second best is in green. The best overall (for each column) is underlined.

			class		file		project		Python Avg.	Java Avg.	Overall Avg.
Method	Venue	Size	python	java	python	java	python	java			
CodeGen (Nijkamp et al., 2022)	ICLR'23	350M	5.82	8.30	7.79	0.00	3.91	6.14	5.84	4.81	5.33
PanGu-Coder (Christopoulou et al., 2022)	ArXiv'22	300M	7.82	19.90	9.41	0.00	6.09	7.95	7.77	9.28	8.53
			Llama3.2-1B-Instruct								
Direct	Meta	1B	5.45	1.00	4.41	0.00	8.70	2.27	6.19	1.09	3.64
RepoCoder (Zhang et al., 2023)	EMNLP'23	1B	29.09	23.00	17.65	0.00	21.74	18.18	22.81	13.73	18.27
RLCoder (Wang et al., 2024)	ICSE'25	1B	27.27	22.00	17.65	0.00	13.04	20.45	19.32	14.15	16.73
Ours (FileRAG)	**This work**	1B	7.27	1.00	1.47	0.00	4.35	0.00	4.36	0.33	2.35
Ours (Infile + FileRAG)	**This work**	1B	3.64	6.00	0.00	0.00	0.00	0.00	1.21	2.0	1.61
			Llama3.2-3B-Instruct								
Direct	Meta	3B	9.09	13.00	11.76	0.00	4.35	4.55	8.40	5.85	7.13
RepoCoder (Zhang et al., 2023)	EMNLP'23	3B	27.27	26.00	17.65	0.00	21.74	18.18	22.22	14.73	18.47
RLCoder (Wang et al., 2024)	ICSE'25	3B	27.27	26.00	16.18	0.00	21.74	18.18	21.73	14.73	18.23
Ours (FileRAG)	**This work**	3B	36.36	32.00	13.24	0.00	13.04	18.18	20.88	16.73	18.80
Ours (Infile + FileRAG)	**This work**	3B	38.18	44.00	17.65	0.00	13.04	25.00	22.96	23.00	22.98
			DeepSeekCoder-7B-Instruct								
Direct	Meta	7B	14.55	8.08	19.12	0.00	8.70	2.27	14.12	3.45	8.79
RepoCoder (Zhang et al., 2023)	EMNLP'23	7B	32.73	32.00	13.24	0.00	8.70	20.45	18.22	17.48	17.85
RLCoder (Wang et al., 2024)	ICSE'25	7B	34.55	30.00	11.76	0.00	8.70	20.45	18.34	16.82	17.58
Ours (FileRAG)	**This work**	7B	12.73	OUT	14.71	OUT	4.35	OUT	10.60	-	-
Ours (Infile + FileRAG)	**This work**	7B	5.45	OUT	4.41	OUT	8.70	OUT	6.19	-	-
			Llama3.1-8B-Instruct								
Direct	Meta	8B	12.73	11.00	11.76	0.00	4.35	9.09	9.60	6.70	8.16
RepoCoder (Zhang et al., 2023)	EMNLP'23	8B	34.55	31.00	14.71	0.00	21.74	20.45	23.67	17.15	20.41
RLCoder (Wang et al., 2024)	ICSE'25	8B	40.00	30.00	19.12	0.00	21.74	25.00	26.95	18.33	22.64
Ours (FileRAG)	**This work**	8B	34.55	46.00	26.47	0.00	17.39	31.82	26.14	25.94	26.04
Ours (Infile + FileRAG)	**This work**	8B	50.91	68.00	26.47	100.00	26.09	45.45	34.49	71.15	52.82

Table 4: Comparison of our best solution with chatGPT (GPT3.5) in CoderEval dataset. The best results is in blue, and the second best is in green.

			class		file		project		Python Avg.	Java Avg.	Overall Avg.
Method	Venue	Size	python	java	python	java	python	java			
			chatGPT								
Direct (GPT-3.5)	OpenAI	175B	8.73	22.40	21.03	0.00	9.57	16.14	13.11	12.85	12.98
ReACC (GPT-3.5) (Lu et al., 2022)	ACL'22	175B	20.36	-	17.65	-	11.30	16.44	-	-	
CoCoGen (GPT-3.5) (Bi et al., 2024)	ArXiv'24	175B	28.00	-	30.29	-	21.30	-	26.53	-	-
RepoCoder (GPT-3.5) (Zhang et al., 2023)	EMNLP'23	175B	35.45	-	29.41	-	16.96	-	27.27	-	-
Self-Collaboration (GPT-3.5) (Dong et al., 2023)	ArXiv'23	175B	21.82	-	20.59	-	13.04	-	18.48	-	-
Ours (FileRAG - Llama3.1-8B-Inst.)	**This work**	8B	34.55	46.00	26.47	0.00	17.39	31.82	26.14	25.94	26.04
Ours (Infile + FileRAG - Llama3.1-8B-Inst.)	**This work**	8B	50.91	68.00	26.47	100.00	26.09	45.45	34.49	71.15	52.82

average Python (13.11 vs. 9.60) and Java (12.85 vs. 6.70) performances. This indicates that solely relying on parametric language model knowledge, GPT-3.5 is better than Llama3.1-8B-Instruct. This shows Llama3.1-8B-Instruct is strongly benefited by our designed setups allowing it to outperform GPT-3.5 and solutions based on it. We did not perform any experiment with GPT-3.5 since we focus on open-source and small-scale language models, and it would require an extra API calling cost that would not be affordable with the project's budget.

5 Conclusion

In this work we aimed to explore different context formation strategies for repository-aware code generation without requiring retriever or generator training, nor compiler feedback. With a comprehensive evaluation of different context formation strategies, impact of retrievers and language models, we provide a deep analysis about how these different pieces influence repository-aware code generation. By validating different hypothesis about how the retrieved functions must be inserted in the language models context, we conclude that its entire scope, i.e. the retrieved function whole file, is the most powerful context. It provides a full function contextualization, where it is implemented, imports, dependencies and even examples about how it behaves. Also, the current file where the new function will be implemented in, if available, also provide a strong guide to code generation, following previous findings (Feng et al., 2024).

In particular, our solution keeps repository safeguards since: 1) we do not require training, avoiding sensitive leakage of proprietary repositories, 2) we do not require running the code repository to obtain compiler feedback for implementation improvement, alleviating the impact of malicious or hallucinated code, 3) Our explorations and solutions are designed for open-source small-scale language models, broadening the application for environments with limited computational resources, and avoids data sharing with proprietary models. Our solution can help software developers and speedup software engineering, and help to understand repository structures.

6 Limitations

While this work presents a comprehensive analysis and demonstrates the effectiveness of specific context strategies for repository-aware code generation, it is essential to acknowledge the boundaries of the study. The following limitations outline the key constraints regarding the practical scalability of our methods, the generalizability of the findings to different programming environments, the scope of the models tested, the chosen generation paradigm, and the dependency on the quality of the underlying codebase.

Overall Scalability and Context Size: A primary limitation of the study is the scalability of its most effective methods, which is directly tied to the significant context size required for generation. The best-performing strategy, "Infile + FileRAG," demands large context windows, averaging around 9K tokens for Python and 17K for Java. This dependency creates practical scalability challenges, especially in projects with extremely large, monolithic files, where including an entire file via "FileRAG" becomes impractical. The authors acknowledge this issue and, as part of their future work, plan to explore context summarization and selection strategies to decrease the context size while preserving performance.

Generalizability Across Languages and Project Types: The study's experiments are confined to Python and Java within the CoderEval benchmark. The effectiveness of the proposed context strategies is not guaranteed to translate to other programming languages (e.g., JavaScript, Rust, C++) or different types of software projects, such as web development, embedded systems, or data science notebooks, which may have different structural conventions.

Limited Scope of Language Models due to Resource Constraints: This study deliberately focused on open-source, small-scale language models to ensure reproducibility and manage resource costs. This choice introduces a limitation, as the effectiveness of our context strategies is highly sensitive to the base model. The performance improvements observed may be smaller on models with different architectural constraints (e.g., smaller context windows) or larger on future, more capable models. Our inability to conduct extensive tests on large-scale proprietary models, such as GPT-3.5, was due to budget and resource limitations, which are common constraints in academic research. Therefore, while the strategies proved potent, their precise impact may vary when applied to different LLMs.

Limited Scope of Generation Paradigm: This study's methodology was scoped to a single generation attempt per task, providing a clear baseline for the effectiveness of each context strategy. This approach, however, does not leverage more advanced, multi-output paradigms. For instance, the use of Self-Consistency (Wang et al., 2022) was not investigated. This is a notable limitation, as such strategies could improve the robustness and accuracy of the generated code. Future work will explore these avenues, particularly through the development of multi-agent systems where different agents can generate and validate solutions.

Dependence on Codebase Quality: The success of the "Infile" and "FileRAG" strategies hinges on the assumption that the existing code in the repository is of high quality. These methods might be less effective or even counterproductive in legacy systems or projects with poorly documented, unstructured, or buggy code. Providing a low-quality file as context could lead the LLM to replicate bad practices or introduce new errors.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Xuanhua Shi, and Hai Jin. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. *ArXiv*, abs/2403.16792.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhong yi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li yu Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jia Wei, and 3 others. 2022. Pangu-coder: Program synthesis with function-level language modeling. *ArXiv*, abs/2207.11280.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*.

Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. 2024. Complexcodeeval: A benchmark for evaluating large code models on more complex code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1895–1906.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Annual Meeting of the Association for Computational Linguistics*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183*.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2021. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118*.

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2023. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *ArXiv*, abs/2203.07722.

Eric Melz. 2023. Enhancing llm intelligence with arm-rag: Auxiliary rationale memory for retrieval augmented generation. *arXiv preprint arXiv:2311.04177*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *ArXiv*, abs/2203.13474.

Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, and 1 others. 1995. Okapi at trec-3. *Nist Special Publication Sp*, 109:109.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems*, 33:16857–16867.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2023a. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing. *arXiv preprint arXiv:2311.09868*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Fengji Zhang, B. Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Conference on Empirical Methods in Natural Language Processing*.

A Full prompts

In this appendix section, we show the exact prompts we employed for code generation for each setup depicted in Table 1. They are depicted from Figure 2 to Figure 9.

B Complementary qualitative results

From Figures 10 to 21 we depict successful and failure cases for each of the three main evaluation setups in CoderEval ("class", "file" and "project"). Our analysis below focus in the "project" setup, which is the most challenging one requiring a repository-level knowledge for successful implementation.

In Figure 18, we depict a successful case for project_runnable sample in Python. We see that the generated implementation (above) is close to the ground truth. The language model is able to correctly execute the steps of the function and is able to call "Response" with exactly the same parameters as expected in the ground truth. In Figure 19 we depict a failure case for one sample in project_level for Python. Despite of being an error, we see that the generated code is very close to the ground truth. All parameters of the "Structure" class, which is implemented in another file in the repository and requires a repository-level understanding to call it, are correctly placed by the language model, matching exactly with the ground truth. Which makes it a failure case are the parameters values. For instance, "months" variable should be zero, but the language models assigns "months = days // 30". This makes the code to fail the test cases. Still, in a practical scenario, our solution has the ability to provide a good starting point to generate functions to be integrated to the repository.

Similar conclusions can be drawn from the Java samples. In Figure 20, our solution enables the language model to generate a more concise code than the ground truth. Differently than the ground truth, the generated code calls the "getContentLengthLong()" method instead of " getContentLength()" as in the ground truth. Still, the generated code passes the test cases. In Figure 21, as happens in Python, our solution provides a code similar to the ground truth, providing a starting point already aligned to the repository implementations, requiring less work and time to adapt it to the expected code.

```
You are an expert developer. You must write a function or method based on the
function signature, docstring and documentation provided in the context of the
function (restate the function signature). Firstly, make a high level plan with
steps that progressively solves the task. Based on this plan, generate the code.
If a method signature is provided, implement just the method, do not implement
the whole class. The generated code must be between [CODE] and [/CODE] tags.

### Context with implementations:
#########
FILE 1
#########

def dehydrate_pandas_timedelta(value):
    """ Dehydrator for `pandas.Timedelta` values.

    :param value:
    :type value: pandas.Timedelta
    :returns:
    """
    return dehydrate_duration(Duration(
        nanoseconds=value.value
    ))



### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.

    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 2: Standard RAG prompt. The function "dehydrate_pandas_timedelta" is retrieved from the vector store and included in the context without further contextualization.

```
You are an expert developer. You must write a function or method based on the
function signature, docstring and documentation provided in the context of the
function (restate the function signature). Firstly, make a high level plan with
steps that progressively solves the task. Based on this plan, generate the code.
If a method signature is provided, implement just the method, do not implement the
whole class. The generated code must be between [CODE] and [/CODE] tags.

### Context with implementations:
#########
FILE 1
#########
import sys

    sys.path.append("/home/travis/builds/repos/neo4j---neo4j-python-driver/")
from datetime import datetime, time, timedelta, timezone
from src.neo4j._optional_deps import np, pd
from src.neo4j.time import Date, DateTime, Duration, MAX_YEAR, MIN_YEAR,
    NANO_SECONDS,
    Time,
)
from src.neo4j._codec.packstream import Structure

ANY_BUILTIN_DATETIME = datetime(1970, 1, 1)
if np is not None:
if pd is not None:
if np is not None:
    _NUMPY_DURATION_UNITS = {"Y": "years", "M": "months", "W": "weeks",
    "D": "days", "h": "hours", "m": "minutes", "s": "seconds",
    "ms": "milliseconds","us": "microseconds","ns": "nanoseconds",}

if pd is not None:
    def dehydrate_pandas_timedelta(value):
        """ Dehydrator for `pandas.Timedelta` values.
        :param value:
        :type value: pandas.Timedelta
        :returns:
        """
        return dehydrate_duration(Duration(
            nanoseconds=value.value
        ))
if __name__ == "__main__":
    isT=True
    try:
        res1 = hydrate_time(3723000000004, 3600)
        res2 = hydrate_time(3723000000004, None)
        if not str(res1)=="01:02:03.000000004+01:00"
        or not str(res2)=="01:02:03.000000004":
            isT=False
    except:
        isT=False
    if not isT:
        raise Exception("Result not True!!!")

### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.

    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 3: The proposed RAG-Statement (RAG-S) prompt. The function "dehydrate_pandas_timedelta" is retrieved from the vector store and included along with import statements, declarations, and other code pieces. All functions and methods with different name from "dehydrate_pandas_timedelta" are removed.

```
You are an expert developer. You must write a function or method based on the
function signature, docstring and implementations provided in the context of the
function (restate the function signature). If a method signature is provided,
implement just the method, do not implement the whole class. The generated code
must be between [CODE] and [/CODE] tags.

### Context with implementations:
#########
FILE 1
#########
def _validate_version(version):

        if isinstance(version, numbers.Number):
            # Some people apparently take "version number" too literally :)
            version = str(version)

        if version is not None:
            try:
                packaging.version.Version(version)
            except (packaging.version.InvalidVersion, TypeError):
                warnings.warn(
                    "The version specified (%r) is an invalid version, this "
                    "may not work as expected with newer versions of "
                    "setuptools, pip, and PyPI. Please see PEP 440 for more "
                    "details." % version
                )
                return setuptools.sic(version)
        return version

#########
FILE 2
#########
def _to_seconds(td):
    '''Convert a timedelta to seconds'''
    return td.seconds + td.days * 24 * 60 * 60

Here are the steps to implement it:
1. **Validate the input**: Check if the input `value` is indeed a timedelta object.
If not, raise a TypeError with a descriptive message.
2. **Extract relevant information**: Extract the days, seconds, and microseconds
from the timedelta object `value` to prepare it for dehydration.

### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.

    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 4: The proposed RAG of Toughts (RAGoT) prompt. The functions "_validate_version" and "_to_seconds" are retrieved from the vector store based on the steps 1 and 2, respectively, present in the prompt.

```
You are an expert developer (... <same text as in previous listings> ...).
The generated code must be between [CODE] and [/CODE] tags.

### Context with implementations:
#########
FILE 1
#########
import io
# ...
# More imports
# ...
from setuptools.extern.packaging import version

class Distribution(_Distribution):
    # ... class docstring ...
    _DISTUTILS_UNSUPPORTED_METADATA = {# ... Dictionary declaration ...}
    _patched_dist = None

    def _validate_version(version):
        if isinstance(version, numbers.Number):
            # Some people apparently take "version number" too literally :)
            version = str(version)
        if version is not None:
            try:
                packaging.version.Version(version)
            except (packaging.version.InvalidVersion, TypeError):
                warnings.warn("The version specified (%r) is an invalid version,
                this ""may not work as expected with newer versions of "
                    "setuptools, pip, and PyPI. Please see PEP 440 for more "
                    "details." % version)
                return setuptools.sic(version)
        return version

#########
FILE 2
#########
'''Base classes and helpers for building zone specific tzinfo classes'''
from datetime import datetime, timedelta, tzinfo
# ...
# More imports
# ...
__all__ = []
# ... global variables declaration ... #
def _to_seconds(td):
    '''Convert a timedelta to seconds'''
    return td.seconds + td.days * 24 * 60 * 60

Here are the steps to implement it:
1. **Validate the input**: Check if the input `value` is indeed a timedelta
object. If not, raise a TypeError with a descriptive message.
2. **Extract relevant information**: Extract the days, seconds, and
microseconds from the timedelta object `value` to prepare it for dehydration.

### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.

    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 5: The proposed RAGoT-Scope (RAGoT-S) prompt. The functions "_validate_version" and "_to_seconds" are retrieved from the vector store based on the steps 1 and 2, respectively, present in the prompt.

```
You are an expert developer (... <same text as in previous listings> ...).
The generated code must be between [CODE] and [/CODE] tags.

### Context with implementations:
#########
FILE 1
#########
# Copyright (c) "Neo4j"
# ... comments about the code license ... #

import sys
# ... more imports ...
from src.neo4j._codec.packstream import Structure

ANY_BUILTIN_DATETIME = datetime(1970, 1, 1)
def get_date_unix_epoch():
    return Date(1970, 1, 1)

def get_date_unix_epoch_ordinal():
    return get_date_unix_epoch().to_ordinal()

                o
                o
                o

# ... <other functions and classes implementations in the file> ... #

                o
                o
                o

if pd is not None:
    def dehydrate_pandas_timedelta(value):
        """ Dehydrator for `pandas.Timedelta` values.
        :param value:
        :type value: pandas.Timedelta
        :returns:
        """
        return dehydrate_duration(Duration( nanoseconds=value.value))

if __name__ == "__main__":
   # ... < main function body > ...

### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.

    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 6: The proposed FileRAG prompt. The function “dehydrate_pandas_timedelta” is retrieved from the vector store, and its entire file is inserted in the context.

```
You are an expert developer (... <same text as in previous listings> ...).
The generated code must be between [CODE] and [/CODE] tags.
### Context with implementations:
#########
FILE 1
#########
import io
# More imports
from setuptools.extern.packaging import version
        o o o
# ... <other functions and classes implementations in the file> ...#
        o o o
class Distribution(_Distribution):
    # ... class docstring ...
    # ... class attributes declaration ...
    def _validate_version(version):
        if isinstance(version, numbers.Number):
            # Some people apparently take "version number" too literally :)
            version = str(version)
        if version is not None:
            try:
                packaging.version.Version(version)
            except (packaging.version.InvalidVersion, TypeError):
                warnings.warn("The version specified (%r) is an invalid version,
                this ""may not work as expected with newer versions of "
                    "setuptools, pip, and PyPI. Please see PEP 440 for more "
                    "details." % version)
                return setuptools.sic(version)
        return version
        o o o
# ... <other functions and classes implementations in the file> ...#
        o o o
#########
FILE 2
#########
'''Base classes and helpers for building zone specific tzinfo classes'''
from datetime import datetime, timedelta, tzinfo
# More imports
# ... global variables declaration ... #
            o o o
# ... <other functions and classes implementations in the file> ...#
            o o o
def _to_seconds(td):
    '''Convert a timedelta to seconds'''
    return td.seconds + td.days * 24 * 60 * 60
           o o o
# ... <other functions and classes implementations in the file> ...#
           o o o
Here are the steps to implement it:
1. **Validate the input**: Check if the input `value` is indeed a timedelta
object. If not, raise a TypeError with a descriptive message.
2. **Extract relevant information**: Extract the days, seconds, and microseconds
from the timedelta object `value` to prepare it for dehydration.

### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.
    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 7: The proposed FileRAGoT prompt. The functions “_validate_version” and “_to_seconds” are retrieved from the vector store based on the steps 1 and 2, respectively, present in the prompt. Their entire files are included in the context.

```
You are an expert developer (... <same text as in previous listings> ...).
The generated code must be between [CODE] and [/CODE] tags.

### Context with implementations:
#########
FILE 1
#########
# Copyright (c) "Neo4j"
# ... comments about the code license ... #

from datetime import datetime, time, timedelta
# ... more imports ...

def get_date_unix_epoch():
    return Date(1970, 1, 1)

def get_date_unix_epoch_ordinal():
    return get_date_unix_epoch().to_ordinal()

def get_datetime_unix_epoch_utc():
    from pytz import utc
    return DateTime(1970, 1, 1, 0, 0, 0, utc)

def hydrate_date(days):
    # ... <function body implementation > ...

def dehydrate_date(value):
    # ... <function body implementation > ...

def hydrate_time(nanoseconds, tz=None):
    # ... <function body implementation > ...

def dehydrate_time(value):
    # ... <function body implementation > ...

def hydrate_datetime(seconds, nanoseconds, tz=None):
    # ... <function body implementation > ...

def dehydrate_datetime(value):
    # ... <function body implementation > ...

def hydrate_duration(months, days, seconds, nanoseconds):
    # ... <function body implementation > ...

def dehydrate_duration(value):
    # ... <function body implementation > ...

### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.
    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 8: The proposed Infile prompt. In this case, there is no retrieval. We just move the function "dehydrate_timedelta" to the end of its file and remove its body. Then we ask the generator to implement based just in its current file implementation.

```
You are an expert developer (... <same text as in previous listings> ...).
The generated code must be between [CODE] and [/CODE] tags.
### Context with implementations:
#########
FILE 1
#########
def dehydrate_pandas_timedelta(value):
    """ Dehydrator for `pandas.Timedelta` values.
        :param value:
        :type value: pandas.Timedelta
        :returns:
        """
        return dehydrate_duration(Duration(nanoseconds=value.value  ))
#########
FILE 2
#########
# Copyright (c) "Neo4j"
# ... comments about the code license ... #

from datetime import datetime, time, timedelta
# ... more imports ...

def get_date_unix_epoch():
    return Date(1970, 1, 1)

def get_date_unix_epoch_ordinal():
    return get_date_unix_epoch().to_ordinal()

def get_datetime_unix_epoch_utc():
    from pytz import utc
    return DateTime(1970, 1, 1, 0, 0, 0, utc)

def hydrate_date(days):
    # ... <function body implementation > ...

def dehydrate_date(value):
    # ... <function body implementation > ...

def hydrate_time(nanoseconds, tz=None):
    # ... <function body implementation > ...

def dehydrate_time(value):
    # ... <function body implementation > ...

def hydrate_datetime(seconds, nanoseconds, tz=None):
    # ... <function body implementation > ...

def dehydrate_datetime(value):
    # ... <function body implementation > ...

def hydrate_duration(months, days, seconds, nanoseconds):
    # ... <function body implementation > ...

def dehydrate_duration(value):
    # ... <function body implementation > ...
### Function docstring:
def dehydrate_timedelta(value): """
    """
    Dehydrator for `timedelta` values.
    :param value:
    :type value: timedelta
    :return:
    """
```

Figure 9: The proposed Infile + RAG prompt. We keep the prompt of Listing 2 along with the one in Listing 8. All other variations, like "Infile + RAGoT", "Infile + fileRAG", etc., follow the same rationale.

```
### Predicted
def values(self, *keys):
    """Return the values of the record,
        optionally filtering to
    include only certain values by index
         or key.

    :param keys: indexes or keys of the
        items to include; if none
                  are provided, all
                      values will be
                      included
    :return: list of values
    :rtype: list
    """
    if keys:
        d = []
        for key in keys:
            try:
                i = self.index(key)
            except KeyError:
                d.append(None)
            else:
                d.append(self[i])
        return d
    return list(self)
```

```
### Ground Truth
def values(self, *keys):
    """ Return the values of the record,
         optionally filtering to
    include only certain values by index
         or key.

    :param keys: indexes or keys of the
        items to include; if none
                  are provided, all
                      values will be
                      included
    :return: list of values
    :rtype: list
    """
    if keys:
        d = []
        for key in keys:
            try:
                i = self.index(key)
            except KeyError:
                d.append(None)
            else:
                d.append(self[i])
        return d
    return list(self)
```

Figure 10: Successful case for a *class_runnable* Python sample employing "Infile + FileRAG" with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
def index(self, key):
    """
    Return the index of the given item.

    :param key: a key
    :return: index
    :rtype: int
    """
    try:
        return self.__keys.index(str(key
            ))
    except ValueError:
        raise IndexError(f"Key '{key}'
            not found in the record")
```

```
### Ground Truth
def index(self, key):
    """ Return the index of the given
        item.

    :param key: a key
    :return: index
    :rtype: int
    """
    if isinstance(key, int):
        if 0 <= key < len(self.__keys):
            return key
        raise IndexError(key)
    elif isinstance(key, str):
        try:
            return self.__keys.index(key
                )
        except ValueError:
            raise KeyError(key)
    else:
        raise TypeError(key)
```

Figure 11: Failure case for a *class_runnable* Python sample employing "Infile + FileRAG" with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
/**
 * Decodes octets to characters using
     the UTF-8 decoding and appends
 * the characters to a StringBuffer.
 * @return the index to the next
     unchecked character in the string
     to decode
 */
private static int decodeOctets(int i,
    ByteBuffer bb, StringBuilder sb) {
    // If there is only one octet and it
        is an ASCII character
    if (bb.limit() == 1 && (bb.get(0) &
        0xFF) < 0x80) {
        // Octet can be appended
            directly
        sb.append((char) bb.get(0));
        return i + 2;
    } else {
        // Decode the octets using UTF-8
             decoding
        CharBuffer cb = UTF_8_CHARSET.
            decode(bb);
        sb.append(cb);
        return i + bb.limit() * 3 - 1;
    }
}
```

```
### Ground Truth
private static int decodeOctets(int i,
    ByteBuffer bb,StringBuilder sb){
  if (bb.limit() == 1 && (bb.get(0) & 0
      xFF) < 0x80) {
     sb.append((char)bb.get(0));
     return i + 2;
  }
 else {
     CharBuffer cb=UTF_8_CHARSET.decode(
         bb);
     sb.append(cb);
     return i + bb.limit() * 3 - 1;
  }
}
```

Figure 12: Successful case for a *class_runnable* Java sample employing “Infile + FileRAG” with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
/**
 * Schedules a file to be deleted when
     JVM exits.
 * If file is directory delete it and
     all sub-directories.
 *
 * @param file  file or directory to
     delete, must not be {@code null}
 * @throws NullPointerException if the
     file is {@code null}
 * @throws IOException in case deletion
     is unsuccessful
 */
public static void forceDeleteOnExit(
    File file) throws IOException {
    if (file == null) {
        throw new NullPointerException("
            File must not be null");
    }
    if (file.exists()) {
        if (file.isDirectory()) {
            deleteDirectoryOnExit(file);
        } else {
            forceDeleteOnExit(file);
        }
    }
}
```

```
### Ground Truth
public static void forceDeleteOnExit(
    File file) throws IOException {
  if (file.isDirectory()) {
    deleteDirectoryOnExit(file);
  }
 else {
    file.deleteOnExit();
  }
}
```

Figure 13: Failure case for a *class_runnable* Java sample employing “Infile + FileRAG” with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
def normalize_cmd(cmd: tuple[str,...]) -> tuple[str,...]:
    """
    Fixes for the following issues on windows
    - https://bugs.python.org/issue8557
    - windows does not parse shebangs

    This function also makes deep-path shebangs work just fine
    """
    # Check if the input cmd is a tuple of strings
    if not isinstance(cmd, tuple) or not all(isinstance(x, str) for x in cmd):
        raise ValueError("Input cmd must be a tuple of strings")

    # If the tuple contains a string that is a full path to an executable, return
        the tuple as is
    for exe in cmd:
        if os.path.isfile(exe) and os.access(exe, os.X_OK):
            return cmd

    # If the tuple contains a string that is a shebang, normalize it to a full path
        by finding the executable using find_executable
    for i, exe in enumerate(cmd):
        if exe.startswith('#!'):
            exe_path = find_executable(exe[2:])
            if exe_path is not None:
                cmd = tuple(x if x!= exe else exe_path for x in cmd)
                break

    # If the tuple contains a string that is not a full path or a shebang, find the
        executable using find_executable and replace the string with the full path
    for i, exe in enumerate(cmd):
        exe_path = find_executable(exe)
        if exe_path is not None:
            cmd = tuple(x if x!= exe else exe_path for x in cmd)
            break

    return cmd
```

```
### Ground Truth
def normalize_cmd(cmd: tuple[str, ...]) -> tuple[str, ...]:
    """Fixes for the following issues on windows
    - https://bugs.python.org/issue8557
    - windows does not parse shebangs

    This function also makes deep-path shebangs work just fine
    """
    # Use PATH to determine the executable
    exe = normexe(cmd[0])

    # Figure out the shebang from the resulting command
    cmd = parse_filename(exe) + (exe,) + cmd[1:]

    # This could have given us back another bare executable
    exe = normexe(cmd[0])

    return (exe,) + cmd[1:]
```

Figure 14: Successful case for a *file_runnable* Python sample employing “Infile + FileRAG” with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
@deprecated("'foo' has been deprecated in favour of 'bar'")
def foo(x):
    pass
```

```
### Ground Truth
def deprecated(message):
    """ Decorator for deprecating functions and methods.

    ::

        @deprecated("'foo' has been deprecated in favour of 'bar'")
        def foo(x):
            pass

    """
    def decorator(f):
        if asyncio.iscoroutinefunction(f):
            @wraps(f)
            async def inner(*args, **kwargs):
                deprecation_warn(message, stack_level=2)
                return await f(*args, **kwargs)

            return inner
        else:
            @wraps(f)
            def inner(*args, **kwargs):
                deprecation_warn(message, stack_level=2)
                return f(*args, **kwargs)

            return inner

    return decorator
```

Figure 15: Failure case for a *file_runnable* Python sample employing “Infile + FileRAG” with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
/**
 * Performs a right node rotation.
 *
 * @param node a node to rotate
 * @return a new parent of the {@code
     node}
 */
private TreeNode<T> rotateRight(TreeNode
    <T> node)
{
    TreeNode<T> left = node.left;
    left.parent = null;

    node.setLeftChild(left.right);
    left.setRightChild(node);

    node.updateHeightAndSubtreeSize();
    left.updateHeightAndSubtreeSize();

    return left;
}
```

```
### Ground Truth
private TreeNode<T> rotateRight(TreeNode
    <T> node){
  TreeNode<T> left=node.left;
  left.parent=null;
  node.setLeftChild(left.right);
  left.setRightChild(node);
  node.updateHeightAndSubtreeSize();
  left.updateHeightAndSubtreeSize();
  return left;
}
```

Figure 16: Successful case for a *file_runnable* Java sample employing "Infile + FileRAG" with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
/**
 * Performs a right node rotation.
 *
 * @param node a node to rotate
 * @return a new parent of the {@code
     node}
 */
private TreeNode<T> rotateRight(TreeNode
    <T> node)
{
    // Update the node's left child to
       be its current right child
    TreeNode<T> newRoot = node.right;
    node.setRightChild(newRoot.left);

    // Update the new root's left child
       to be the node's current left
       child
    newRoot.setLeftChild(node);

    // Update the node's parent to be
       the new root
    node.parent = newRoot;

    // Update the new root's parent to
       be the node's current parent
    newRoot.parent = node.parent;

    // Update the new root's right child
        to be the node's current right
       child
    newRoot.right = node.right;

    // Update the new root's height and
       subtree size
    newRoot.updateHeightAndSubtreeSize()
        ;

    // Return the new parent of the
       rotated node
    return newRoot;
}
```

```
### Ground Truth
private TreeNode<T> rotateRight(TreeNode
    <T> node){
  TreeNode<T> left=node.left;
  left.parent=null;
  node.setLeftChild(left.right);
  left.setRightChild(node);
  node.updateHeightAndSubtreeSize();
  left.updateHeightAndSubtreeSize();
  return left;
}
```

Figure 17: Failure case for a *file_runnable* Java sample employing "FileRAG" with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```python
### Predicted
def discard(self, n=-1, qid=-1, dehydration_hooks=None, hydration_hooks=None, **
    handlers):
    """ Appends a DISCARD message to the output queue.

    :param n: number of records to discard, default = -1 (ALL)
    :param qid: query ID to discard for, default = -1 (last query)
    :param dehydration_hooks:
        Hooks to dehydrate types (dict from type (class) to dehydration
        function). Dehydration functions receive the value and returns an
        object of type understood by packstream.
    :param hydration_hooks:
        Hooks to hydrate types (mapping from type (class) to
        dehydration function). Dehydration functions receive the value of
        type understood by packstream and are free to return anything.
    :param handlers: handler functions passed into the returned Response object
    """
    log.debug("[#%04X]  C: DISCARD %r", self.local_port, (n, qid))
    self._append(b"\x14", (n, qid),
                 Response(self, "discard", hydration_hooks, **handlers),
                 dehydration_hooks=dehydration_hooks)
```

```python
### Ground Truth
def discard(self, n=-1, qid=-1, dehydration_hooks=None,
            hydration_hooks=None, **handlers):
        # Just ignore n and qid, it is not supported in the Bolt 3 Protocol.
        log.debug("[#%04X]  C: DISCARD_ALL", self.local_port)
        self._append(b"\x2F", (),
                     Response(self, "discard", hydration_hooks, **handlers),
                     dehydration_hooks=dehydration_hooks)
```

Figure 18: Successful case for a *project_runnable* Python sample employing “Infile + FileRAG” with Llama3.1-8B-Isntruct. Predicted is above and the ground-truth below.

```
### Predicted
def dehydrate_timedelta(value):
    """ Dehydrator for `timedelta` values.

    :param value:
    :type value: timedelta
    :return:
    """
    days, seconds = value.days, value.seconds
    months = days // 30
    days %= 30
    nanoseconds = (seconds + value.microseconds // 1000000) * 1000000000
    return Structure(b"E", months, days, seconds, nanoseconds)
```

```
### Ground Truth
def dehydrate_timedelta(value):
    """ Dehydrator for `timedelta` values.

    :param value:
    :type value: timedelta
    :return:
    """
    months = 0
    days = value.days
    seconds = value.seconds
    nanoseconds = 1000 * value.microseconds
    return Structure(b"E", months, days, seconds, nanoseconds)
```

Figure 19: Failure case for a *project_runnable* Python sample employing "Infile + FileRAG" with Llama3.1-8B-Isntruct. Predicted is above and the ground-truth below.

```
### Predicted
public long contentLength() {
    return request.getContentLengthLong();
}
```

```
### Ground Truth
public long contentLength(){
  long size;
  try {
    size=Long.parseLong(request.getHeader(FileUpload.CONTENT_LENGTH));
  }
 catch (  NumberFormatException e) {
    size=request.getContentLength();
  }
  return size;
}
```

Figure 20: Successful case for a *project_runnable* Java sample employing "Infile + FileRAG" with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.

```
### Predicted
/**
     Produces a formatted string as specified by the conversion pattern.
  */
public String format(LoggingEvent event) {
    StringBuffer sbuf = new StringBuffer(BUF_SIZE);
    PatternConverter head = createPatternParser(getConversionPattern()).parse();
    head.format(sbuf, event);
    return sbuf.toString();
}
```

```
### Ground Truth
public String format(LoggingEvent event){
  if (sbuf.capacity() > MAX_CAPACITY) {
    sbuf=new StringBuffer(BUF_SIZE);
  }
 else {
    sbuf.setLength(0);
  }
  PatternConverter c=head;
  while (c != null) {
    c.format(sbuf,event);
    c=c.next;
  }
  return sbuf.toString();
}
```

Figure 21: Failure case for a *project_runnable* Java sample employing “Infile + FileRAG” with Llama3.1-8B-Instruct. Predicted is above and the ground-truth below.