RAR: Retrieval-augmented retrieval for code generation in low-resource languages

Avik Dutta
Microsoft
Bangalore, India

Mukul Singh
Sumit Gulwani
Vu Le
Microsoft
Redmond, US

Gust Verbruggen
Microsoft
Keerbergen, Belgium

Abstract

Language models struggle in generating code for low-resource programming languages, since these are underrepresented in training data. Either examples or documentation are commonly used for improved code generation. We propose to use both types of information together and present retrieval augmented retrieval (RAR) as a two-step method for selecting relevant examples and documentation. Experiments on three low-resource languages (Power Query M, OfficeScript and Excel formulas) show that RAR outperforms independent example and grammar retrieval (+2.81–26.14%). Interestingly, we show that two-step retrieval selects better examples and documentation when used independently as well.

1 Introduction

Large language models (LLMs) struggle to generate low-resource programming languages from natural language due to limited pre-training knowledge (Luo et al., 2023; Wang et al., 2023b; Singh et al., 2023). Previous work improves low-resource code generation with LLMs using retrieval augmented generation (RAG) with relevant examples (Poesia et al., 2022; Khatry et al., 2023c) or documentation (Zhou et al., 2022; Ma et al., 2024).

There are several challenges in using documentation as context for code generation. First, documentation often does not include how the components are actually pieced together in the form of real code, which makes it difficult for models to understand the syntax and usage for new languages. Second, documentation is weakly correlated to the natural language utterance and specific parts of documentation might not be related at all to the utterance but crucial for the code generation. For example, the Power Query M code to *"Highlight top 5 projects based on sales"* requires a flag `OrderByDescending` to be set, but this is not related to the utterance. Third, documentation is often dense and flat, and selecting the right subset of documentation is both challenging and crucial to the success of these systems.

Similarly, using examples poses some additional challenges. First, obtaining an example bank of code and natural language pairs requires a lot of manual effort. Second, models tend to overfit to examples provided in the prompt and do not properly adapt them to the current problem.

To address these challenges, we propose Retrieval Augmented Retrieval (RAR) for code generation from both examples and documentation. The examples provide specific information and the documentation allows the model to generalize that specific information to the current problem. Our approach enhances the retrieval process by leveraging the outputs of an initial retriever, the *driver retriever*, to guide a secondary retriever, the *influenced retriever*. This sequential retrieval mechanism ensures relatedness of the retrieved examples and grammar entities, which is required for the model to specialize and generalize accordingly.

We evaluate RAR on three low-resource languages: (Power Query) M, Excel formulas (Excel) and OfficeScript (OS). We compare dependent example or documentation selection of RAR to multiple existing documentation and example retrieval techniques, showing improvements of $+25\%$ on OS and $+3\%$ on M for documentation; and $+1.28\%$ for OS, $+3.5\%$ for M and $+2\%$ on Excel for examples. When combining examples and documentation, RAR shows improvements of $+4\%$ on OS, $+2\%$ on M and $+4\%$ on Excel over independent retrieval. We also analyze the impact of using two-step retrieval, including only relevant and irrelevant context items in the prompt, and the token length.

We make the following contributions:

- We use a two-step retrieval method, where the influenced retriever leverages the findings and mistakes of the driver retriever.

- We demonstrate that publicly available documentation is sufficient for NL-to-Code generation tasks, even for low-resource languages.
- We show that retrieval-augmented retrieval (RAR) works better for selecting either examples, documentation, or both, when compared to independent retrieval.

2 Related Work

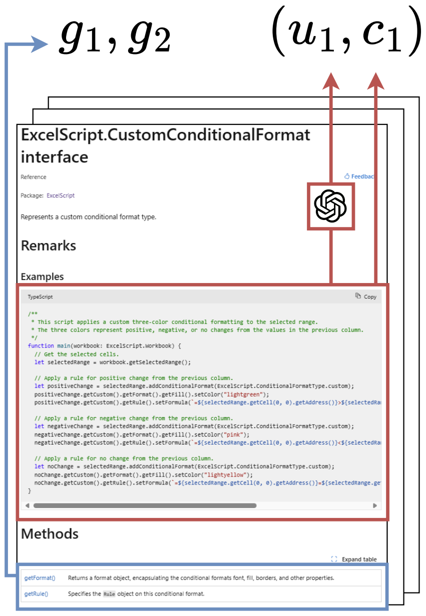
Multiple techniques have been developed to improve code generation from natural language with LLMs, including (1) retrieval augmented generation for adding contextually relevant examples to the prompt (Khatry et al., 2023c; Poesia et al., 2022; Khatry et al., 2023a); (2) execution-guided refinement (Kroening et al., 2004; Chen et al., 2019); and (3) reasoning involving chain-of-thought variants adapted to programming tasks (Li et al., 2023; Le et al., 2024).

These techniques often struggle in generating accurate generations for low-resource programming languages. Recent work focused on code generation for low resource languages has leveraged documentation as context instead of examples (Bareiß et al., 2022; Zhou et al., 2022). CAPIR is one such popular technique, which uses contextually relevant parts of the documentation as inputs to code models. Grammar prompting (Wang et al., 2023a) also follows this paradigm. One drawback of these techniques is that documentation, even though complete, often does not provide the same signals to the models as examples. Documentation nodes that do not seem semantically aligned with the task also tend to be ignored.

3 Documentation and examples

Documentation is the most comprehensive and structured resource (Roehm et al., 2012) publicly available (Forward and Lethbridge, 2002) for most programming languages. The documentation consists of a grammar (D) that describes how code is built over entities (classes, methods, properties) and examples (E) that depicts how to use and combine elements from D. An example of grammar and examples from a page of the OfficeScript documentation is shown in Figure 1.

The **grammar** (D) serves as a bank for grammar elements over which retrieval is performed. We consider each grammar element g_i to be one



ExcelScript.CustomConditionalFormat interface

Reference

Feedback

Package: ExcelScript

Represents a custom conditional format type.

Remarks

Examples

TypeScript

Copy

```
/**
 * This script applies a custom three-color conditional formatting to the selected range.
 * The three colors represent positive, negative, or no changes from the values in the previous column.
 */
function main(workbook: ExcelScript.Workbook) {
  // Get the selected cells.
  let selectedRange = workbook.getSelectedRange();

  // Apply a rule for positive change from the previous column.
  let positiveChange = selectedRange.addConditionalFormat(ExcelScript.ConditionalFormatType.custom);
  positiveChange.getCustom().getFormat().getFill().setColor("lightgreen");
  positiveChange.getCustom().getRule().setFormula(`=${selectedRange.getCell(0, 0).getAddress()}>${selectedRa

  // Apply a rule for negative change from the previous column.
  let negativeChange = selectedRange.addConditionalFormat(ExcelScript.ConditionalFormatType.custom);
  negativeChange.getCustom().getFormat().getFill().setColor("pink");
  negativeChange.getCustom().getRule().setFormula(`=${selectedRange.getCell(0, 0).getAddress()}<${selectedRa

  // Apply a rule for no change from the previous column.
  let noChange = selectedRange.addConditionalFormat(ExcelScript.ConditionalFormatType.custom);
  noChange.getCustom().getFormat().getFill().setColor("lightyellow");
  noChange.getCustom().getRule().setFormula(`=${selectedRange.getCell(0, 0).getAddress()}=${selectedRange.ge
}
```

Methods

Expand table

getFormat()	Returns a format object, encapsulating the conditional formats font, fill, borders, and other properties.
getRule()	Specifies the `Rule` object on this conditional format.

Figure 1: Illustrates how we extract the grammar (blue marker) and examples (red marker) from the publicly available documentation to build their respective corpora for retrieval.

standalone function or class method. We can use the path of each node in an abstract syntax tree (AST) to extract elements $\in D$ from a snippet of code. For example, even if multiple classes have a method `getFormat`, then following a path up the AST allows us to disambiguate which class this method is from. A full example of code and the associated grammar entities is shown in Figure 2.

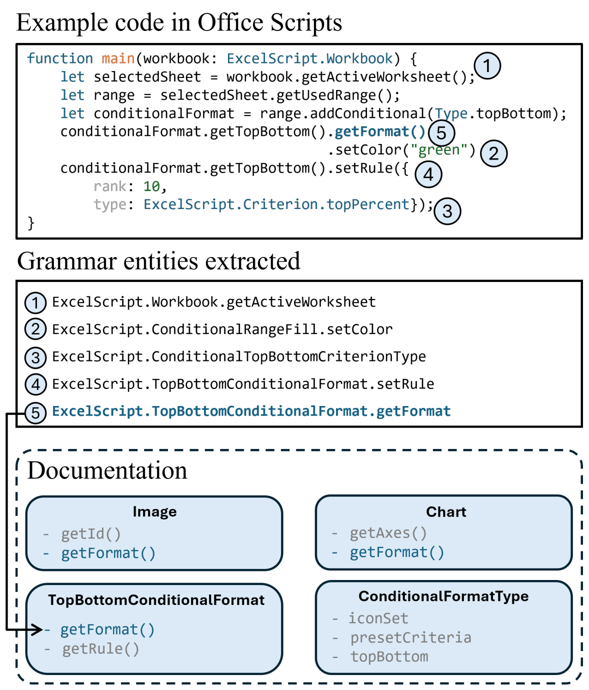
The **example corpus** (E) is composed of description (utterance) and code pairs (u_i, c_i). We only consider examples present in the documentation, which consists of sample code illustrating the usage of grammar elements. If a textual description of an example is not available, we use an LLM (`gpt-4`) to generate it.

4 Retrieval Augmented Retrieval

RAR uses a two-step retrieval where the driver retriever (R_D) influences the influenced retriever (R_I). There are two possible scenarios:

1. **Example → grammar**: R_D retrieves from E and R_I retrieves from D.
2. **Grammar → example**: R_D retrieves from D and R_I retrieves from E.

We first describe how to use and fine-tune embeddings for retrieval, and then show how this is applied in R_D and R_I in both these scenarios.


Example code in Office Scripts

```
function main(workbook: ExcelScript.Workbook) {  (1)
    let selectedSheet = workbook.getActiveWorksheet();
    let range = selectedSheet.getUsedRange();
    let conditionalFormat = range.addConditional(Type.topBottom);
    conditionalFormat.getTopBottom().getFormat() (5)
                                     .setColor("green") (2)
    conditionalFormat.getTopBottom().setRule({ (4)
        rank: 10,
        type: ExcelScript.Criterion.topPercent}); (3)
}
```

Grammar entities extracted

```
(1) ExcelScript.Workbook.getActiveWorksheet
(2) ExcelScript.ConditionalRangeFill.setColor
(3) ExcelScript.ConditionalTopBottomCriterionType
(4) ExcelScript.TopBottomConditionalFormat.setRule
(5) ExcelScript.TopBottomConditionalFormat.getFormat
```

Figure 2: Example code entities (1 to 5) extracted from a sample OfficeScript program. The extracted entities are mapped to grammar nodes using the abstract syntax tree of the node. (5) in figure is mapped to `TopBottomConditionalFormat` despite the same property being present in `Image` and `Chart`.

4.1 Embeddings for retrieval

Retrieval commonly relies on an embedding cosine similarity

$$S_{\mathcal{M}}(s_1, s_2) = S_C(\mathcal{M}(s_1), \mathcal{M}(s_2))$$

with $\mathcal{M}$: `string` $\rightarrow \mathbb{R}^n$ the embedding model used to map a `string` onto an n-dimensional vector and S_C the cosine similarity.

Off-the-shelf embedding models struggle to link user utterances and grammar elements. To counter this, we fine-tune $\mathcal{M}^*$ to predict whether a grammar element g is used in the code c associated with utterance u. In other words, the model is trained on

$$\|S_{\mathcal{M}^*}(u, g) - [g \in c]\|$$

to make the embedding of an utterance u similar to the embedding of grammar element g if g is used in the code associated with u. To select negative labels, we find grammar entities g that are not used in code c, but which are close to c according to $S_{\mathcal{M}}(u, g_n)$ (with the pre-trained model). We also select an equal number of grammar entities that are used in c but have the lowest similarities $S_{\mathcal{M}}(u, g)$.

4.2 Examples → grammar

In this setup, we retrieve examples first and then use it to retrieve the grammar elements.

First, the driver retriever R_D extracts the best k examples (E_k) based on similarity $S_{\mathcal{M}}(u, u_i)$ with u the target utterance. Second, the influenced retriever R_I uses the selected examples E_k to select relevant n grammar elements D_n. Grammar elements are selected in three ways:

- All grammar elements used in $(u, c) \in E_k$ (positively dependent).
- Grammar elements g with high similarity $S_{\mathcal{M}^*}(u, g)$ according to the fine-tuned embedding model $\mathcal{M}^*$ (independent).
- Grammar elements g with high similarity $S_i = S_{\mathcal{M}^*}(u, g)$ **and** low similarity $S_j = S_{\mathcal{M}^*}(u', g)$ to the already selected examples $(u', c') \in E_k$ (negatively dependent). This case considers the possibility that the selected examples were irrelevant. We combine both as $(1 - S_i) + \lambda_{E\rightarrow D}(1 + S_j)$ with $\lambda_{E\rightarrow D}$ a hyperparameter.

4.3 Grammar → examples

In this setup, we first retrieve the grammar entities and then use them to select examples.

First, the driver retriever R_D extracts best n grammar entities (D_n) based on similarity $S_{\mathcal{M}^*}(u, g)$ with u the target utterance. Second, the influenced retriever R_I uses these entities D_n to select examples in two ways.

- With $g(c)$ the grammar elements used in c and $\mathit{idf}(g)$ the inverse document frequency of g with respect to E, we compute a score

  $$\frac{1}{|g(c)|} \sum_{g \in g(c) \cap d_n} \mathit{idf}(g) \cdot S_{\mathcal{M}^*}(u, g)$$

  that measures how relevant an example (u, c) is to the selected D_n (positively dependent). Scaling by idf lets us focus on unique grammar elements.
- Similarly, we can compute the score S_b over $g(c) \cap (D \setminus D_n)$ to consider the case where D_n has irrelevant elements (negatively dependent). Examples are then selected according to $S_b + \lambda_{D\rightarrow E} S_{\mathcal{M}}(u, u')$ with $\lambda_{D\rightarrow E}$ another hyperparameter.

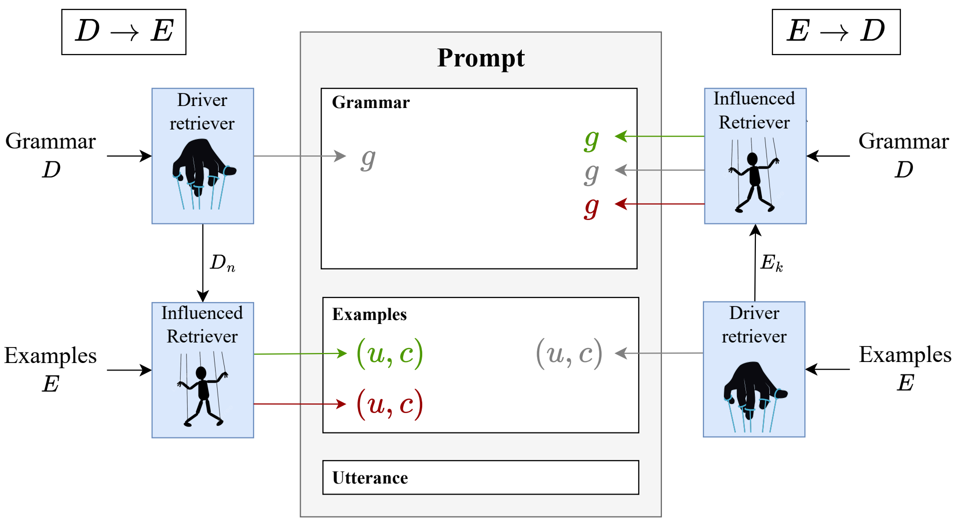


Figure 3: Overview of RAR. ($G \to E$) The driver retriever independently (gray) selects grammar elements and passes them to the influenced retriever, which uses them to select positively (green) and negatively (red) influenced examples. ($E \to G$) The driver retriever independently selects examples and passes them to the influenced retriever, which uses them to select positively (green) and negatively (red) influenced grammar elements, as well as independent (gray) grammar elements.

5 Experimental Setup

We describe the experimental setup and the conditions set for a fair comparison between our approach and the baseline.

5.1 Datasets

We focus our experiments on three low-resource programming languages: **OfficeScript**, **(Power Query) M** and **Excel formulas**.

OfficeScript We use the InstructExcel benchmarks (Payan et al., 2023) and filter them for conditional formatting specific tasks, as we can compute execution match for them (Singh et al., 2022). Examples and grammar are scraped from its documentation.[1]

Power Query M We use the test split of benchmarks used to evaluate TST^R (Khatry et al., 2023b). Besides utterance and code, each test contains a table to execute the code over, which is also provided in the prompt. We scrape the examples and grammar from the official documentation.[2]

Excel We use an existing dataset of derived-column formulas (Singh et al., 2024). Each test contains a spreadsheet table over which the formula is executed. Examples and grammar are scraped from its official documentation. [3]

Dataset	**n**	$\|E\|$	$\|D\|$
Office Scripts	589	17	275
Power Query M	77	144	746
Excel formulas	100	663	505

Table 1: Summary of the datasets: n *implies* dataset size, $|E|$ *implies* #examples, $|D|$ *implies* #doc pages. We extract E and D from documentation which forms the corpora for our approach.

5.2 Metrics

We use *sketch* and *execution* match as metrics for all datasets.

OfficeScript For sketch match, we compute an edit similarity that ignores constants, variable values or function arguments. For execution match, we compare the formatted cells obtained after executing the OfficeScript code.

M For sketch match we mask the constants and identifiers and compute the character-level edit similarity. We compute execution match by executing the programs on the table and checking for table equality.

[1]https://github.com/OfficeDev/office-scripts-docs-reference

[2]https://github.com/OfficeDev/office-js-docs-reference

[3]https://support.microsoft.com/en-us/office/excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188

Model	OfficeScript		M		Formulas	
	Sketch	Execution	Sketch	Execution	Sketch	Execution
$\mathcal{M}_{PT}$	52.28	40.81	65.34	45.28	**75.00**	41.00
$\mathcal{M}_{FT}$	55.99	44.35	56.43	43.40	74.00	40.00
DocPrompting	50.17	38.68	73.64	50.94	**75.00**	41.00
CAPIR	51.69	41.06	71.07	55.68	70.00	**44.00**
RAR$_D$	**86.68**	**70.49**	**74.27**	**58.49**	**75.00**	**44.00**

Table 2: Comparing RAR$_D$ against other **grammar** retrieval methods. Only grammar entities are used for helping the LLM. Each value denotes match accuracy in % (higher is better)

Excel For sketch match we mask cell references and constants and compute the character-level edit similarity. We compute execution match by executing the formula on all rows in the table and checking for equality of the resulting column.

5.3 Baselines and Versions

We define different variations of independent (Ret) and dependent (RAR) retrieval.

- Ret$_D$ uses $\mathcal{M}^*$ to retrieve D_n from D. Only D_n is included in the prompt.
- Ret$_E$ uses $\mathcal{M}$ to retrieve E_k from E. Only E_k is included in the prompt.
- Ret$_{E\perp D}$ uses $\mathcal{M}^*$ to retrieve D_n from D and $\mathcal{M}$ to retrieve E_k from E. Both D_n and E_k are included in the prompt.
- RAR$_D$: R_D operates on E to give E_k and R_I on D to give D_n. Only D_n is included in the prompt.
- RAR$_E$: R_D operates on D to give D_n and R_I on E to give E_k. Only E_k is included in the prompt.
- RAR$_{E\rightarrow D}$: R_D operates on E to give E_k and R_I on D to give D_n. Both E_k and D_n are included in the prompt.
- RAR$_{D\rightarrow E}$: R_D operates on D to give D_n and R_I on E to give E_k. Both D_n and E_k are included in the prompt.

5.4 Models

We use `text-embedding-ada-002` as the pre-trained embedding model $\mathcal{M}_{PT}$ and SentenceBERT (Reimers and Gurevych, 2019) for $\mathcal{M}_{FT}$. We use GPT-4 (32K) (Brown et al., 2020) as the base LLM.

6 Evaluation

We aim to answer the following research questions:

RQ1 How does dependent retrieval for documentation or examples compare against existing (independent) documentation and example retrieval methods?

RQ2 Does dependent retrieval extract better context than standalone retrieval of examples and documentation?

RQ3 Are both positively and negatively dependent retrieval necessary?

RQ4 How does the performance vary as a function of increasing context length?

RQ5 Does RAR rely on the driver retriever for its performance gain over independent retrievers?

6.1 Dependent documentation or example retrieval (RQ1)

We evaluate RAR against other retrieval methods over both examples and documentation. We use a fixed number of retrieved examples and grammar entities for each task. For OfficeScript, we extract 3 examples and 66 grammar entities. For M and Formulas, we extract 10 examples and 20 grammar entities.

6.1.1 Baselines

For grammar retrieval, we consider (1) retrieval by calculating cosine similarity of pre-trained embedding model ($\mathcal{M}$), (2) retrieval by calculating cosine similarity of fine-tuned embedding model ($\mathcal{M}^*$), (3) **DocPrompting** (Zhou et al., 2022), which uses BM25 retriever, (4) **CAPIR** (Ma et al., 2024), which is a divide-and-conquer and re-ranking based strategy for retrieval.

Models	OfficeScript		M		Formulas	
	Sketch	Execution	Sketch	Execution	Sketch	Execution
$\mathcal{M}_{PT}$	83.42	69.04	74.24	50.94	74.00	40.00
TST	64.76	52.95	70.21	51.16	**75.00**	42.00
TST^R	73.86	60.37	69.90	45.35	**75.00**	44.00
RAR$_E$	**85.67**	**70.32**	**76.29**	**54.72**	**75.00**	**46.00**

Table 3: Comparing RAR against other **example** retrieval techniques. Only methods are used here for helping the LLM. Each value denotes match accuracy in % (higher is better)

For example retrieval, we consider (1) $\mathcal{M}$, (2) **TST** (Poesia et al., 2022) and **TST**^{R} (Khatry et al., 2023c), which fine-tune SentenceBERT and a small dense network on top of $\mathcal{M}$ to make utterance intents reflect their respective code similarities.

6.1.2 Results

Table 2 and Table 3 show that RAR outperform the baselines for both grammar and example retrieval. RAR$_D$ shows significant gain in grammar extraction for OfficeScript. It has an execution match gain of 26.14% against the best performing baseline ($\mathcal{M}$ with SentenceBERT). For M, we see an execution match gain of 2.81% and 2% for Excel over CAPIR.

We find RAR$_E$ to retrieve better examples to aid code generation. The respective baselines cover both pre-trained and fine-tuned (TST and TST^R) versions of retrieval. Our dependent retrieval strategy performs better than either case. For M, we find the improvement in both sketch and execution match to be marginal. This implies that the grammar elements retrieved by R_D are able to guide the extraction of relevant examples for those utterances, which were difficult to extract using direct similarity of embeddings.

We note that the fine-tuned models TST and TST^R perform worse than the unsupervised embedding model $\mathcal{M}_{PT}$. We attribute this to the fact that our training set is only scraped from documentation and thus smaller with low variations of the same function, and fine-tuning can more easily overfit.

6.2 Dependent documentation and example retrieval (RQ2)

We compare our dependent approach with baselines that operate independently on documentation and examples.

6.2.1 Setup

For OfficeScript, we extract $n \approx 66$ grammar entities and $k = 3$ examples. We extract same number of positively and negatively dependent examples. We tune hyper-parameter $\lambda_{E\to D} = 20$ and $\lambda_{D\to E} = 10$.

For M, we extract $n \approx 20$ grammar entities and $k = 10$ examples. RAR$_{E\to D}$ uses all positively influenced grammar entities (≈ 10). Setting $\lambda_{E\to D} = 100$, extracting 10 negatively influenced grammar elements for each example and de-duplicating them yields another ≈ 10 grammar elements. We set $\lambda_{D\to E} = 10$ for RAR$_{D\to E}$.

For Excel, we retrieve $n \approx 20$ grammar entities and $k = 10$ examples. We set $\lambda_{E\to D} = 10$ for RAR$_{E\to D}$, where we approximately take 10 negatively dependent grammar elements out of the final 20. For RAR$_{D\to E}$, we take $\lambda_{D\to E} = 10$ and extract an equal split of positively and negatively impacted examples.

6.2.2 Results

Table 4 shows that dependent retrieval (RAR) consistently performs better than independent retrieval (Ret) even if only a single type of context is provided.

Grammar Independent retrieval of grammar performs significantly worse than retrieving grammar through relevant examples (-25% for OfficeScript, -15% for M and -4% for Excel). This shows that RAR is able to pick more relevant documentation, without requiring examples to show how they should be used in the context of a program.

Examples When independently retrieving examples, the difference between RAR and independent retrieval is smaller. Still, RAR consistently performs better. On M, retrieving only examples using RAR achieves the highest sketch match, indicating the similarity of the retrieved examples.

		Office Scripts		M		Excel	
Context	Method	Sketch	Exec	Sketch	Exec	Sketch	Exec
G	Ret$_D$	55.99	44.35	56.43	43.40	74.00	40.00
	RAR$_D$	86.68	70.49	74.27	58.49	75.00	44.00
E	Ret$_E$	83.42	69.04	74.24	50.94	74.00	40.00
	RAR$_E$	85.67	70.32	**76.29**	54.72	75.00	**46.00**
G + E	Ret$_{E \perp D}$	87.18	72.34	73.40	58.49	**76.00**	38.00
	RAR$_{E \to D}$	**92.36**	**76.40**	72.87	58.49	71.00	42.00
	RAR$_{D \to E}$	90.71	76.01	74.86	**60.38**	75.00	39.00

Table 4: Comparison of RAR with independent retrieval techniques. Context implies whether only grammar (D), or examples (E), or both ($D + E$) have been included in the prompt for LLM. Methods with *Ret* are the independent retrievers with the subscript defining their corpus. The values denote match accuracy in % (higher the better). RAR outperforms its Ret counterpart for all context scenarios.

Grammar and examples Grammar + examples together yields better results than separate (+6% for OfficeScript and +2% for M). The examples help the model in figuring out the general program structure, and the documentation helps in figuring out how to adapt these examples. This is highlighted in M where sketch match is highest when only using examples (+1.5% over RAR$_{D \to E}$), but execution match is significantly higher for the latter (+5%). For Excel, we found that examples alone gave better results than used along with grammar. Grammar adds bias to necessarily use functions for queries which can be solved without them, eg. `=[@Class]&": "&[@Assignment]`. But grammar includes `CONCATENATE` function in the prompt which motivates the LLM to use it, eventually giving a different join of texts than what was expected. Barring such scenarios, grammar helped with different overloading of formula functions when used along with examples.

Recall in grammar Table 5 reports the recall of retrieving relevant grammar entities for Ret$_D$ and RAR$_D$. RAR beats independent retrieval again with a considerable margin (+25% for OfficeScript, +46% for M and +55% for Excel). The relevance of grammar extracted using RAR$_D$ further explains the jump in performance in Table 4.

6.3 Positive and negative influence (RQ3)

We evaluate whether the positive and negative influence assumption of the driver retriever helps us select better context. In this setting, for *influenced* retriever, we consider including only positive ($+$) or only negatively ($-$) influenced elements in the prompt. We compare them with our proposed approach where we use an equivalent count of positively and negatively influenced elements. The number of examples and grammar is kept constant across all scenarios for a fair comparison. Table 6 shows the results.

Method	OfficeScript	M	Excel
Ret$_D$	50.04	24.83	16.53
RAR$_D$	**76.36**	**67.16**	**82.86**

Table 5: Comparison of retrieval quality when grammar is extracted either independently or with RAR. We evaluate quality by taking average of recall Rate (in %) for the occurrence of the retrieved grammar entity in the actual code for comparison.

We find that combining positive and negative elements, based on the result of the driver retriever, output helps in obtaining better context. The retrieval of D_- or E_- is able to catch some important context which get missed when we trust the driver output to be good. For OfficeScript, we see a clear improvement in performance. However, for M, we find that sketch match is better for D_- and E_-, while execution is better for D_+ and E_+. Using both in equal proportion helps us attain a balance when trying to improve both metrics.

6.4 Variation with token size (RQ4)

We vary the token size by changing the number of retrieved examples and grammar entities in the prompt. We use independent retrievers with the same context count as RAR as baseline.

Figure 4 shows that RAR performs better at most

RAR	**R$_I$**	**Office Scripts**		**M**		**Excel**	
		Sketch	**Exec**	**Sketch**	**Exec**	**Sketch**	**Exec**
$E \rightarrow D$	D_+	79.80	64.01	72.38	58.49	68.00	39.00
	D_-	83.47	69.98	74.22	52.83	73.00	37.00
	D_++D_-	92.36	76.40	72.87	58.49	71.00	42.00
$D \rightarrow E$	E_+	88.87	73.19	72.72	64.15	70.00	41.00
	E_-	72.51	58.68	75.70	58.49	76.00	40.00
	E_++E_-	90.71	76.01	74.86	60.38	75.00	39.00

Table 6: Ablation to show importance of assuming R_D output to be both good and bad while retrieving for R_I. In this setting, both example and grammar is used for grounding the LLM. We find clear majority for Office Scripts. For M, we need both positive (+) and negative (−) influence to attain a balanced performance improvement in both metrics.

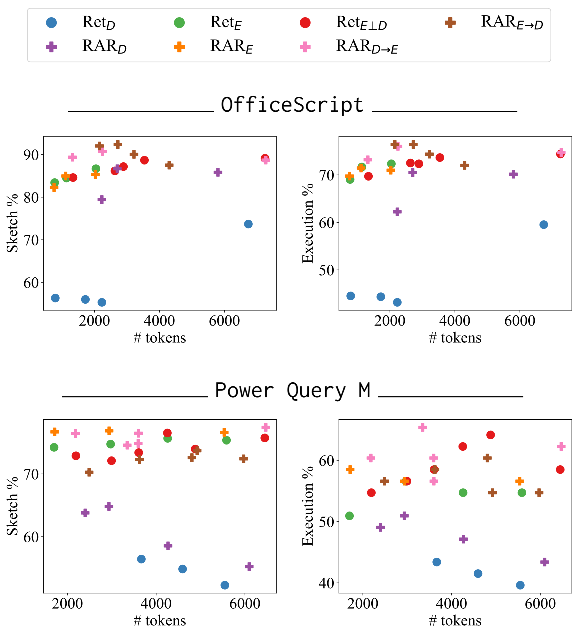
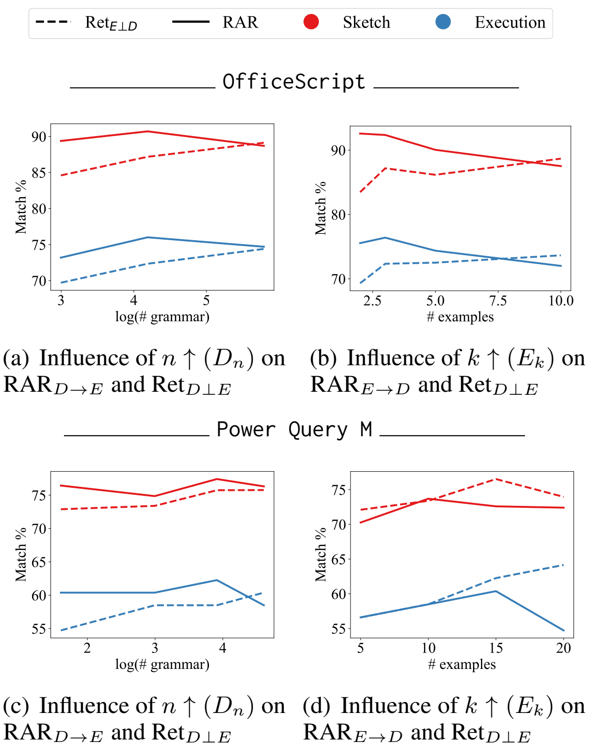


Figure 4: Performance as a function of increasing prompt token length for different approaches. Plots on the left show *sketch* match accuracy and on the right show *execution* match accuracy. RAR outperforms its baseline even for large token sizes. We find lower token lengths are enough for accurate code generation.

token counts. The only exception remains with Ret$_{E\rightarrow D}$ for M, where we find both sketch and execution below baseline for larger token sizes This happens because M has a larger example corpus compared to OfficeScript. The influenced, even while considering driver retriever output to be bad, might be extracting functions from the same pool of incorrect intent. As a result, the performance is low. Moreover, even though the context size increases, the performance remains steady and does not increase further. This removes the notion of models trying to populate the prompt with more content rather than including only the relevant ones. We find that the best context is achieved around the ~3000 token size mark. Further additions simply confuse (can be seen by a slight drop) or play no role in improving the quality of generation.

6.5 Reliance on driver (RQ5)



(a) Influence of $n \uparrow (D_n)$ on RAR$_{D\rightarrow E}$ and Ret$_{D\perp E}$

(b) Influence of $k \uparrow (E_k)$ on RAR$_{E\rightarrow D}$ and Ret$_{D\perp E}$

(c) Influence of $n \uparrow (D_n)$ on RAR$_{D\rightarrow E}$ and Ret$_{D\perp E}$

(d) Influence of $k \uparrow (E_k)$ on RAR$_{E\rightarrow D}$ and Ret$_{D\perp E}$

Figure 5: The impact on performance when the retrieved context size from *driver* is increased. Both the baseline and RAR in each setting have the same R_D output. The only thing which brings a performance difference is the output from R_I. This shows that R_I is not entirely reliant on R_D. It adapts itself to keep the performance above baseline with increasing context length.

Method	Embedding	OfficeScript		M		Excel	
		Sketch	Exec	Sketch	Exec	Sketch	Exec
Ret$_{E\perp D}$	pre-trained	87.86	71.67	74.29	54.72	74.00	38.00
RAR$_{D\to E}$	pre-trained	**88.17**	**73.48**	**75.52**	**54.72**	**77.00**	**39.00**
Ret$_{E\perp D}$	fine-tuned	87.18	72.34	73.4	58.49	**76.00**	38.00
RAR$_{D\to E}$	fine-tuned	**90.71**	**76.01**	**74.86**	**60.38**	75.00	**39.00**

Table 7: Shows that our approach performs better than the baseline even when different embeddings for retrieval is used. This further consolidates that R_I is independent of R_D and can even improve performance with other retrieval styles.

We compare independent retrievers Ret$_{E\perp D}$ with RAR$_{D\to E}$ and RAR$_{E\to D}$, by altering the *driver* retrieval (1) output size, and (2) method. This helps us understand how R_I behaves as R_D changes. Including R_D's output in the prompt also enables us to understand the impact of R_I alone as we compare with the baseline, containing the same R_D output. This provides a clear view on the impact R_I has towards performance improvement.

Increasing driver output We find in Figure 5 that RAR is better than its baseline when both example and grammar retrieved from the driver is increased. There is a general trend of the match accuracy declining as we increase the output size. This implies that R_I is unable to infer a specific topic from R_D's output to make a positive or negative influence assumption. So the retrieval becomes randomized, and it fails to converge to a particular topic for a candidate solution. We also find the performance for RAR going below its baseline for M in $E \to D$ setting.

When R_I retrieves grammar from an increasing number of extracted examples, more similar grammar elements are retrieved (like `CONCATENATE` and `CONCAT` in Excel). The LLM now receives multiple grammar elements very similar to the utterance, which causes confusion while choosing the right ones. On the other hand, independent retrieval extracts a more diverse grammar, because it does not overfit on the examples. This makes identifying the right grammar elements easier and hence results in a better match numbers when compared with RAR.

Altering retrieval method We compare Ret$_{E\perp D}$ with RAR$_{D\to E}$ for two different settings: using pre-trained embeddings $\mathcal{M}$ and using fine-tuned embeddings $\mathcal{M}^*$ (SentenceBERT) for retrieval. Table 7 shows that RAR still holds its ground and performs better than the baselines even when the retrieval style for R_D is changed. The positive and negatively influenced retrieval assumption helps the influenced retriever to adapt to the changing driver, and eventually fetches context which is relevant to the solution. This shows that our approach can adapt and perform well even with other driver retrievers.

7 Conclusion

We introduce RAR, a two-step retrieval technique used to extract relevant context for code generation over low-resource programming languages. Our approach claims that off-the-shelf documentation for a language is enough to help an LLM generate syntactically and semantically correct programs. We also show how grammar and example work better together. Our approach establishes a working relationship between the two, capable of generating sound and reliable programs. The results we outline opens gates for future research, where grammar and example complement each other to formulate unseen programming languages.

8 Limitations and ethical considerations

Despite showing that RAR performs best at different token counts, combining both grammar and examples significantly increases the number of tokens and thus cost. Our method relies on extensive documentation, which might not be available for all low-resource languages.

We only scrape public documentation that is openly accessible. We do not use any unethical methods to extract data from sources that are protected by privacy policies.

References

Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *Preprint*, arXiv:2206.01335.

Tom Brown, Mann, Ryder, Subbiah, Kaplan, Dhariwal, Neelakantan, Shyam, Sastry, Askell, Agarwal, Herbert-Voss, Krueger, Henighan, Child, Ramesh, Ziegler, Wu, Winter, Hesse, Chen, Sigler, Litwin, Gray, Chess, Clark, Berner, McCandlish, Radford, Sutskever, and Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

Andrew Forward and Timothy C. Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 26–33, New York, NY, USA. Association for Computing Machinery.

Anirudh Khatry, Yasharth Bajpai, Priyanshu Gupta, Sumit Gulwani, and Ashish Tiwari. 2023a. Augmented embeddings for custom retrievals. *Preprint*, arXiv:2310.05380.

Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avrilia Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. 2023b. From words to code: Harnessing data for program synthesis from natural language. *Preprint*, arXiv:2305.01598.

Anirudh Khatry, Sumit Gulwani, Priyanshu Gupta, Vu Le, Ananya Singha, Mukul Singh, and Gust Verbruggen. 2023c. Tstr: Target similarity tuning meets the real world. In *Findings of EMNLP 2023*. Association for Computational Linguistics.

Daniel Kroening, Alex Groce, and Edmund Clarke. 2004. Counterexample guided abstraction refinement via program execution. In *Formal Methods and Software Engineering*, pages 224–238, Berlin, Heidelberg. Springer Berlin Heidelberg.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *Preprint*, arXiv:2310.08992.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional api recommendation for library-oriented code generation. *ArXiv*, abs/2402.19431.

Justin Payan, Swaroop Mishra, Mukul Singh, Carina Negreanu, Christian Poelitz, Chitta Baral, Subhro Roy, Rasika Chakravarthy, Benjamin Van Durme, and Elnaz Nouri. 2023. Instructexcel: A benchmark for natural language instruction in excel. *Preprint*, arXiv:2310.14495.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *ArXiv*, abs/2201.11227.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, Mohammad Raza, and Gust Verbruggen. 2022. Cornet: A neurosymbolic approach to learning conditional table formatting rules by example. *arXiv preprint arXiv:2208.06032*.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. Codefusion: A pre-trained diffusion model for code generation. *Preprint*, arXiv:2310.17680.

Usneek Singh, José Cambronero, Sumit Gulwani, Aditya Kanade, Anirudh Khatry, Vu Le, Mukul Singh, and Gust Verbruggen. 2024. An empirical study of validating synthetic data for formula generation. *Preprint*, arXiv:2407.10657.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023a. Grammar prompting for domain-specific language generation with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 65030–65055. Curran Associates, Inc.

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

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv:2207.05987*.