Guiding Language Models of Code with Global Context using Monitors

Lakshya A Agrawal
Microsoft Research
Bangalore, India
t-lakagrawal@microsoft.com

Aditya Kanade
Microsoft Research
Bangalore, India
kanadeaditya@microsoft.com

Navin Goyal
Microsoft Research
Bangalore, India
navingo@microsoft.com

Shuvendu K. Lahiri
Microsoft Research
Redmond, United States
shuvendu.lahiri@microsoft.com

Sriram K. Rajamani
Microsoft Research
Bangalore, India
sriram@microsoft.com

Abstract

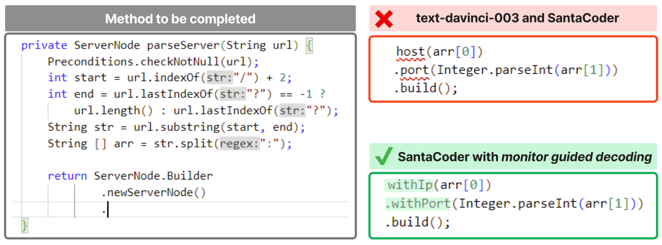
Language models of code (LMs) work well when the surrounding code in the vicinity of generation provides sufficient context. This is not true when it becomes necessary to use types or functionality defined in another module or library, especially those not seen during training. LMs suffer from limited awareness of such global context and end up hallucinating, e.g., using types defined in other files incorrectly. Recent work tries to overcome this issue by retrieving global information to augment the local context. However, this bloats the prompt or requires architecture modifications and additional training.

Integrated development environments (IDEs) assist developers by bringing the global context at their fingertips using static analysis. We extend this assistance, enjoyed by developers, to the LMs. We propose a notion of *monitors* that use static analysis in the background to guide the decoding. Unlike *a priori* retrieval, static analysis is invoked iteratively during the entire decoding process, providing the most relevant suggestions on demand. We demonstrate the usefulness of our proposal by monitoring for type-consistent use of identifiers whenever an LM generates code for object dereference.

We curate PRAGMATICCODE, a dataset of real-world open-source Java projects complete with their development environments and dependencies. Additionally, we curate DOTPROMPTS, a method-level completion task created from prompts in PRAGMATICCODE for evaluation.

On models of varying parameter scale, we show that *monitor-guided decoding* consistently improves the ability of an LM to not only generate identifiers that match the ground truth but also improves compilation rates and agreement with ground truth. Further, we find that LMs with fewer parameters, when guided with our monitor, can outperform larger LMs. With monitor-guided decoding, SantaCoder-1.1B achieves better compilation rate and next-identifier match than the much larger text-davinci-003 model. The datasets and code will be released at `https://aka.ms/monitors4codegen`.

Preprint. Under review.

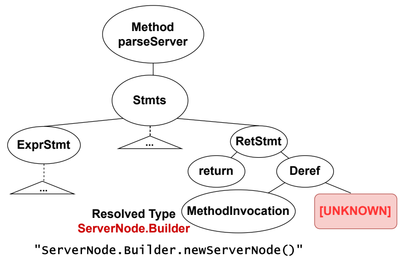

Method to be completed
text-davinci-003 and SantaCoder
SantaCoder with *monitor guided decoding*

```
private ServerNode parseServer(String url) {
    Preconditions.checkNotNull(url);
    int start = url.indexOf(str:"/") + 2;
    int end = url.lastIndexOf(str:"?") == -1 ?
        url.length() : url.lastIndexOf(str:"?");
    String str = url.substring(start, end);
    String [] arr = str.split(regex:":");

    return ServerNode.Builder
            .newServerNode()
            .
}
```

```
host(arr[0])
.port(Integer.parseInt(arr[1]))
.build();
```

```
withIp(arr[0])
.withPort(Integer.parseInt(arr[1]))
.build();
```

(a) Example where text-davinci-003 and SantaCoder generate wrong identifiers, but SantaCoder with monitor-guided decoding generates correct identifiers.



(b) Annotated partial AST for the code to the left.

Figure 1: Motivating example to illustrate monitor-guided decoding.

1 Introduction

Language models of code (LMs), such as in Chen et al. (2021); Nijkamp et al. (2023); Allal et al. (2023) and many others, are revolutionizing code generation. Even commercial offerings based on LMs, like GitHub Copilot, Amazon Code Whisperer and Replit, are now available. The LMs work well when the surrounding code in the vicinity of generation provides sufficient context. This condition does not hold when it becomes necessary to use types or functionality defined in another module or library, especially those not seen during training. In the absence of awareness of such global context, the LMs end up hallucinating, e.g., using types defined in other files incorrectly.

As an example, consider the partial code (Method to be completed) in Figure 1(a). To complete this code, an LM has to generate identifiers consistent with the type of the object returned by `ServerNode.Builder.newServerNode()`. The method `newServerNode` and its return type, class `ServerNode.Builder`, are defined in another file. If an LM does not have information about the `ServerNode.Builder` type, it ends up hallucinating.

We show a completion generated by the OpenAI text-davinci-003 (Ouyang et al., 2022) and SantaCoder (Allal et al., 2023) models in the box decorated with ✖ in Figure 1(a). The completion uses identifiers `host` and `port`, which do not exist in the type `ServerNode.Builder`. The generated code therefore results in "symbol not found" compilation errors.

Integrated development environments (IDEs) have long been at the forefront of delivering assistance to developers. Due to their popularity, LMs are being integrated into IDEs, e.g., the Codex model is used by GitHub Copilot in VSCode. Surprisingly, the other way is under-explored, that is, the use of years of expert tooling in IDEs with LMs. Our inspiration is the use of static analysis by IDEs to bring the global context at the fingertips of developers. Many analyses are integrated in IDEs (Fuhrer, 2013) that help infer and enforce semantic constraints on the code under development. These include resolving def-use, symbol references, type hierarchies and so on. In this work, we focus on the type-directed code completion analysis available in popular IDEs to provide guidance to an LM.

We propose a notion of *monitors* as a stateful interface between LMs and static analysis. A monitor observes the code generated by an LM and invokes static analysis at pre-defined trigger points. The suggestions returned by the static analysis are converted to masks which are used for reshaping the logits (or equivalently, token-generation probabilities) produced by the LM in the subsequent decoding steps. We call our method *monitor-guided decoding* (MGD). Unlike an LM, a static analysis operates on the entire repository and its dependencies. While the LM generates completions by conditioning on the local context, the static analysis ensures consistency with the rest of the code in the repository. Through MGD, we bring the two together without any changes to the LM.

Figure 1(a) also shows the code generated by the SantaCoder model with MGD in the box decorated with ✓. This code makes use of identifiers that are actually defined in the class `ServerNode.Builder`. It compiles and matches the ground truth. In comparison, the *same* SantaCoder model without MGD generates the erroneous code shown in the box decorated with ✖.

Some recent approaches use static analysis (Shrivastava et al., 2022; Ding et al., 2022; Pei et al., 2023) or retrieval (Zhang et al., 2023) to extract relevant code fragments from the global context. These approaches expand the prompt (Shrivastava et al., 2022; Pei et al., 2023; Zhang et al., 2023) or require architecture modifications (Ding et al., 2022) and additional training (Ding et al., 2022; Pei et al., 2023). In comparison, we provide token-level guidance to a frozen LM by invoking static analysis on demand. Our method is complementary to these approaches as they condition the generation by modifying the *input* to the LM, whereas we apply *output*-side constraints by reshaping the logits.

We make the following contributions in this paper:

- We propose *monitor guided decoding (MGD)* as a stateful interface between LMs and static analysis. A monitor watches the LM generating code, invokes static analysis in the background, and uses the information from the static analysis to effectively guide the decoding stage of LMs.
- We contribute PRAGMATICCODE, a dataset of code repositories, to evaluate approaches (such as MGD) that aim to improve quality of code generated by LMs.
- We instantiate the MGD approach to the problem of generating type-consistent identifiers following an object dereference, using a type-based static analysis of partial code.
- On models of varying parameter scale, we show that MGD consistently improves the ability of an LM to generate type-consistent identifiers, and improves compilation rates and agreement with ground truth. Further, we find that LMs with fewer parameters, in conjunction with MGD, can outperform larger LMs. For instance, with MGD, SantaCoder-1.1B achieves better compilation rate and next-identifier match than the much larger text-davinci-003 model, when both models have a budget of 1 generation each. With a budget of 4 generations, it also surpasses agreement with ground truth of text-davinci-003. We also evaluate MGD with different prompt augmentation and decoding strategies, and analyze effect of complexity of identifier names.

We will release our code and datasets publicly[1].

2 Monitor-Guided Decoding

Background. Static analysis of code (Nielson et al., 2015) is used widely in industry in various applications such as in detecting bugs and optimizing code. While analysis is usually performed on complete code, IDEs have long applied static analysis on incomplete code under development (Reps et al., 1983; Dagenais & Hendren, 2008), using algorithms for incremental parsing and semantic analysis (e.g., type inference) of partial code (Hedin, 1992; Wagner, 1997; Maddox III, 1997).

A key abstraction used in analysis of partial code is partial abstract syntax trees (ASTs) with special nodes to indicate incomplete parts of code. These ASTs are further decorated by semantic information through attribute grammars (Reps et al., 1983) that decorate each AST node with attributes that capture the static semantics not captured in the context-free grammar (such as consistency of types among expressions in an assignment). This can range from computing the type-hierarchy for object oriented languages, binding the variables in AST nodes to their declarations, resolving the types of expressions as well as computing the def-use relationships for resolved AST nodes (Fuhrer, 2013).

Figure 1(b) shows the partial AST for the incomplete code in Figure 1(a). All the statements upto the incomplete `return` statement are completely parsed and subtrees corresponding to them are constructed. Due to space limitations, they are not shown in detail. The subtree for the `return` statement includes a node `[UNKNOWN]` indicating the incomplete part. As shown in Figure 1(b), an incremental semantic analysis resolves the type of the partial expression `ServerNode.Builder.newServerNode()` to `ServerNode.Builder`. Later, we show how to use this type information to construct a monitor, which can then be used to guide an LM to generate type-consistent identifier completions.

While we present results for type-consistent identifier completions in this paper, the approach of using static analysis, interfacing with a monitor, and using the monitor to guide the LM, is general. We can extend this approach to *type-state checking* to ensure that certain APIs are invoked in a manner consistent with ordering rules encoded using state machines (Strom & Yemini, 1986; Fink et al., 2008). For example, we can enforce calling `hasNext` on an `Iterator` variable before calling `next`. Thus, the use of monitors as an interface between static analysis and guided decoding of LMs can be a powerful framework to guide LMs with various semantic analyses in a light-weight manner.

[1]https://aka.ms/monitors4codegen

Basic Concepts and Notation. Consider an LM L_θ operating on a vocabulary V. Let $x_1, \ldots, x_n$ be the *partial code* that has been generated by the LM and x_{n+1} be a candidate next token. Though a vanilla (auto-regressive) prompt would consist only of $x_1, \ldots, x_n$, today many approaches augment it with additional information. We use p to indicate this *additional prompt*, e.g., the suffix information used in fill-in-the-middle prompting (Donahue et al., 2020; Fried et al., 2022; Bavarian et al., 2022).

A *property* φ specifies the constraints that a piece of code needs to satisfy. A *static analysis* A_φ checks whether the partial code satisfies φ. If yes, it returns suggestions to extend it so that the extended code continues to satisfy φ. The static analysis works on the *repository-level context* C which not only includes code spread across multiple files in the repository, but also external dependencies and intermediate artifacts (e.g., code bindings) generated during the build process. Such repository-level context is often very large, diverse and complex. Directly including it as an input to the LM will result in bloating and pass the burden of distilling useful information from it to the LM.

A *monitor* M_φ for a property φ is a tuple $(A_\varphi, s_0, S, \texttt{pre}, \texttt{update}, \texttt{maskgen})$. The monitor starts in the *wait state* s_0. If the partial code satisfies the *pre-condition* `pre` then the monitor is triggered and it invokes A_φ on the partial code. The monitor maintains the suggestions returned by A_φ in its state and uses them to guide sampling of the next token x_{n+1}. Let S be the set of states and $s, s' \in S$ respectively be the current and next states of the monitor. With each sampled token x_{n+1}, it *updates* its state using a function $\texttt{update}(s, x_{n+1})$ which tracks the residual suggestions in s'. When the suggestions are exhausted, it reverts to the wait state s_0. We explain the function `maskgen` below.

Decoding Process. Usually, the next token x_{n+1} can be any token from the vocabulary V, sampled based on the logits ℓ determined by the LM. Unlike the usual decoding, in *monitor-guided decoding*, we supervise the code generation using a monitor M_φ for a property φ. We denote the composition of L_θ and M_φ by $L_\theta || M_\varphi$, meaning that both the LM and the monitor are running concurrently and sampling the tokens jointly. The decoding is conditioned on the partial code $x_1, \ldots, x_n$, the (repository-level) context C, the prompt p and the current state s of the monitor.

Eq. (1) states that whenever the monitor is in the wait state s_0, we sample x_{n+1} as per the logits ℓ determined by the LM (Eq. (2)). Otherwise, the logits are combined with a mask m using a function $\oplus$ such that if $m[x] = 0$ then $\ell[x]$ is reset to a large negative value $-K$ and is left unchanged otherwise. This mask is computed by the function `maskgen` in Eq. (3) guided by the current state s of the monitor. Eq. (4) defines how the state of the monitor evolves. When the pre-condition $\texttt{pre}(s; x_1, \ldots, x_n)$ evaluates to true, the next state s' of the monitor is determined by the suggestions returned by the static analysis A_φ. Otherwise, it is determined by the `update` function.

$$(L_\theta || M_\varphi)(x_{n+1} | x_1, \ldots, x_n; C, p, s) = \begin{cases} \texttt{softmax}(\ell)[X_{n+1}] & \text{if } s = s_0 \text{ is the wait state} \\ \texttt{softmax}(\ell \oplus m)[X_{n+1}] & \text{otherwise} \end{cases} \tag{1}$$

$$\ell = L_\theta(\,\cdot\,| x_1, \ldots, x_n; p) \tag{2}$$

$$m = \texttt{maskgen}(s, V) \tag{3}$$

$$s' = \begin{cases} A_\varphi(x_1, \ldots, x_n; C) & \text{if } s = s_0 \wedge \texttt{pre}(s; x_1, \ldots, x_n) \\ \texttt{update}(s, x_{n+1}) & \text{otherwise} \end{cases} \tag{4}$$

The specifics of the monitor state, and the `pre`, `update` and `maskgen` functions depend on the static analysis A_φ used by the monitor. Our formulation is general and even allows combining multiple static analyses by taking a product of the state-spaces of their respective monitors. In the following, we discuss a specific instantiation of this framework of monitor-guided decoding.

Monitoring for Use of Type-Consistent Identifiers. When an object `obj` of a type T is dereferenced, the next token (or more generally, the sequence of subtokens) should refer to an identifier of a field or method defined by the type T. It can otherwise result in a "symbol not found" error. The type T could be defined in another package, imported file or in a library. Unless T comes from a popular library seen during training, the LM may not have knowledge about T.

Our monitor M_φ is triggered when the partial code $x_1, \ldots, x_n$ ends with a partial object dereference expression "`obj.`" where "`.`" is the dereference operation. This is the pre-condition `pre` we use. We employ a static analysis A_φ which returns all the type-consistent identifiers that can be referenced through `obj`. For this, A_φ performs a global analysis over the partial code, imported files, libraries used, and class hierarchies to infer the type T of `obj` and the identifiers accessible through T. The set of type-consistent identifiers returned by A_φ forms the state of the monitor (see Eq. (4)).

Given a state s and the vocabulary V of the LM, `maskgen` identifies all the tokens in V that are consistent with the suggestions in s. The identifiers returned by the static analysis are tokens as per the programming language, whereas the vocabulary V may use its own space of subtokens (Schuster & Nakajima, 2012; Kudo & Richardson, 2018).The mask m (Eq. (3)) is generated by string matching. For all tokens $t \in V$ that form prefixes of the identifiers in s, the mask value $m[t]$ is set to 1, indicating that they can be sampled. Let E be the set of special symbols that indicate end of an identifier name, e.g., the symbol '`(`' in '`getName(`' or '`,`' in '`x,`'. Let w be a string in s and Σ be the set of all possible characters. If a token $t \in V$ matches the regular expression $w \cdot E \cdot \Sigma^*$ then its mask value $m[t]$ is also set to 1. For all other tokens t in V, the mask value $m[t]$ is set to 0.

Let x_{n+1} be the next token sampled according to the second equation in Eq. (1). If x_{n+1} contains a symbol from E, indicating that a complete identifier name has been generated in accordance with the set returned by A_φ, the monitor reverts to the wait state to wait for the next trigger. Otherwise, the token x_{n+1} must be a prefix of a member of s. The `update` function removes the members in s that are not prefixed by x_{n+1}, and those prefixed by x_{n+1} are updated by pruning the prefix string x_{n+1}. The resulting set of character strings forms the next state s' (see the second equation in Eq. (4)). If A_φ returns an empty set to start with, we abandon the current run. Figure 4 (see Appendix A) shows the interaction between the LM and the monitor for the example in Figure 1.

3 Experimental Setup

Dataset Creation. In order to evaluate MGD, we need real-world repositories with their build environments and dependencies, since the whole point of MGD is to use static analysis to get repository-level context. Most published datasets are standalone, with the exception of CoderEval (Yu et al., 2023) and PyEnvs (Pei et al., 2023), both of which are not publicly available at the time of this writing. Hence we curated PRAGMATICCODE, a dataset of real-world open-source Java projects complete with their development environments and dependencies. We ensure that these repositories were released publicly only after the determined training dataset cutoff date (31 March 2022) for the CodeGen (Nijkamp et al., 2023), SantaCoder (Allal et al., 2023), and text-davinci-003 (Ouyang et al., 2022) family of models, which we use to evaluate MGD. In addition, we create a set of test cases called DOTPROMPTS from PRAGMATICCODE, to specifically measure the effectiveness of MGD when instantiated to using monitors for using type-consistent idenfitiers. Each test case in DOTPROMPTS consists of a prompt to a dereference location (using the "." operator in Java) within a target method, and the task is to complete the remainder of the method. We specifically picked classes that have at least one cross-file dependency, and limited including no more than 20 (randomly chosen) methods from each repository, to ensure diversity in the dataset. Overall, PRAGMATICCODE consists of **151** repositories, and DOTPROMPTS consists of **1924** methods and **14234** dereference prompts. Appendix B gives further details.

Models. We study the effect of performing MGD on code generation with the HuggingFace Transformers (Wolf et al., 2020) implementation of Salesforce CodeGen family of models (CodeGen-{350M, 2B, 6B}-Multi, abbreviated as **CG-{350M, 2B, 6B}** hereafter) (Nijkamp et al., 2023) and BigCode SantaCoder-1.1B (**SC** or SantaCoder hereafter) (Allal et al., 2023). Further, we evaluate OpenAI text-davinci-003 (**TD-3** hereafter), deployed in the Azure OpenAI service as a baseline.

Prompting Strategies. We study the effect of different prompt augmentation techniques when combined with MGD: (1) **Standard**: We include the local file content up to the dereference point and truncate from left to fit the prompt budget. (2) **classExprTypes**: For a given target method belonging to a class `C`, we identify the type of all expressions occurring in `C` (after masking out the target method to prevent leakage). We assign a budget of 20% tokens of total prompt budget to classExprTypes. We include the concatenated file contents for the type definitions of all the identified files and truncate from the left as necessary. (3) **RLPG**: We use the prompt augmentation technique proposed in Shrivastava et al. (2022). We use their released source code and model checkpoints to adapt RLPG to DOTPROMPTS.

Decoding Strategies. We experiment with two decoding strategies: (1) **Autoregressive:** for left-to-right decoding. (2) **Fill-in-the-middle:** We use fill-in-the middle (FIM) setting implemented in SantaCoder (Allal et al., 2023).

Metrics. We use the following metrics to measure the quality of generated code: (1) **Compilation Rate (CR):** We replace the ground truth method with the generated method in the context of the

Table 1: Summary of results with a budget of 6 generations per model: The numbers in parentheses are relative improvements of the "-MGD" configuration over the respective base model.

Config \ (Metric, score@6)	CR	NIM	ISM	PM
CG-350M	51.90	76.18	31.41	24.22
CG-350M-MGD	**64.27 (23.84%)**	**83.08 (9.06%)**	**33.89 (7.91%)**	**25.84 (6.68%)**
CG-2B	56.73	80.29	36.12	28.13
CG-2B-MGD	**70.08 (23.53%)**	**86.57 (7.83%)**	**38.75 (7.30%)**	**29.92 (6.36%)**
CG-6B	57.43	81.03	36.96	28.79
CG-6B-MGD	**71.11 (23.82%)**	**86.94 (7.29%)**	**39.42 (6.67%)**	**30.56 (6.13%)**
SC	58.77	82.32	38.09	29.39
SC-classExprTypes	64.09	85.12	39.92	30.83
SC-RLPG	64.66	85.06	42.27	30.47
SC-FIM	66.40	85.18	42.13	31.22
SC-FIM-classExprTypes	70.02	86.96	42.99	31.81
SC-MGD	**72.09 (22.67%)**	**88.27 (7.22%)**	**40.81 (7.16%)**	**31.35 (6.69%)**
SC-classExprTypes-MGD	**74.79 (16.68%)**	**89.48 (5.13%)**	**41.85 (4.85%)**	**32.17 (4.36%)**
SC-RLPG-MGD	**76.60 (18.46%)**	**89.70 (5.45%)**	**44.41 (5.07%)**	**32.07 (5.25%)**
SC-FIM-MGD	**78.58 (18.33%)**	**89.81 (5.44%)**	**44.44 (5.49%)**	**32.68 (4.68%)**
SC-FIM-classExprTypes-MGD	**79.60 (13.69%)**	**90.38 (3.93%)**	**44.58 (3.70%)**	**33.15 (4.21%)**
TD-3	61.69	85.84	45.08	35.9

complete repository and invoke a clean build. We assign a score of 1 if the compilation succeeds, and 0 otherwise. (2) **Match with the ground truth:** We use three specific metrics to measure how closely the generation matches ground truth, namely (a): **Next Identifier Match (NIM):** If the first Java token generated by the LM matches with the ground truth, we assign a score of 1, 0 otherwise; (b) **Identifier Sequence Match (ISM):** Longest prefix match between the ordered set of identifier names in the ground truth and generated completion, normalized by the number of identifiers in the ground truth; and (c) **Prefix Match (PM):** Longest prefix match between the ordered set of Java tokens (as obtained from a Java Lexer) between the ground truth and generated completion, normalized by the number of tokens in the ground truth.

The implementation details and hyperparameters are discussed in Appendix C.

For our experiments, we use $n = 6$ independent trials. For a budget of $k \in [1, n]$ samples, we compute the aggregate score $score@k$ (see Appendix D). On the discrete-valued metrics (CR and NIM), it is identical to $pass@k, n$ (Chen et al., 2021) which estimates the expected number of successes in k chances. Similarly, on the real-valued metrics (ISM and PM), it estimates the expectation of the maximum value of the corresponding metric given k chances.

4 Evaluation

Table 1 shows the summary of the results for all of our experiments and metrics. For the "-MGD" configurations, we also report the relative improvement over the *base model* in parentheses, where the base model is the same model configuration without MGD. Below, we present a detailed evaluation.

4.1 Effect of MGD on Models across Parameter Scale and Architectures

We present results for all the models on Standard prompts described in Section 3.

Compilation Rate. As shown in Figure 2a, all the base models with MGD, including the smallest model CodeGen-350M for $k \geq 4$, outperform the largest model text-davinci-003, by maximum relative margin of 17% achieved by SantaCoder. All the models with MGD outperform their respective base models on Compilation Rate, by a relative improvement ranging between 23%-24%.

Next Identifier Match. As seen in Figure 2b, all the models with MGD outperform their respective base models, showing a relative improvement ranging between 7.2%-9.1%. The smallest model CodeGen-350M with MGD outperforms the much larger CodeGen-6B with a relative improvement of 2.5% over it. SantaCoder with MGD outperforms the larger CodeGen-6B by a relative margin of 8.9%, and even the largest model text-davinci-003 by 2.8%.

Identifier Sequence Match. Figure 2c shows that all the models with MGD outperform their respective base models on ISM, showing a relative improvement ranging between 6.7%-7.9%.

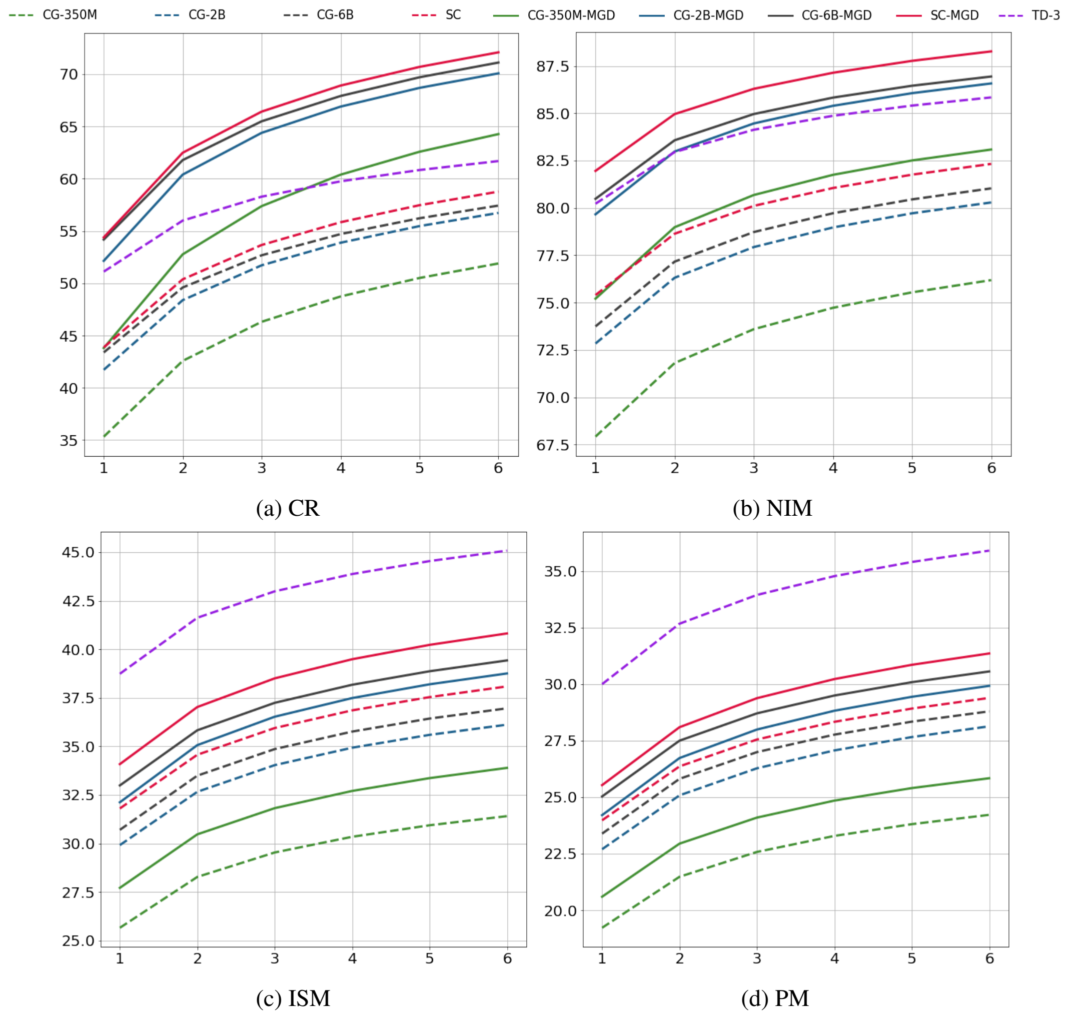


Figure 2: score@k for models with MGD and Standard prompt compared against base models. The values of $k \in [1, 6]$ are marked on the X-axis.

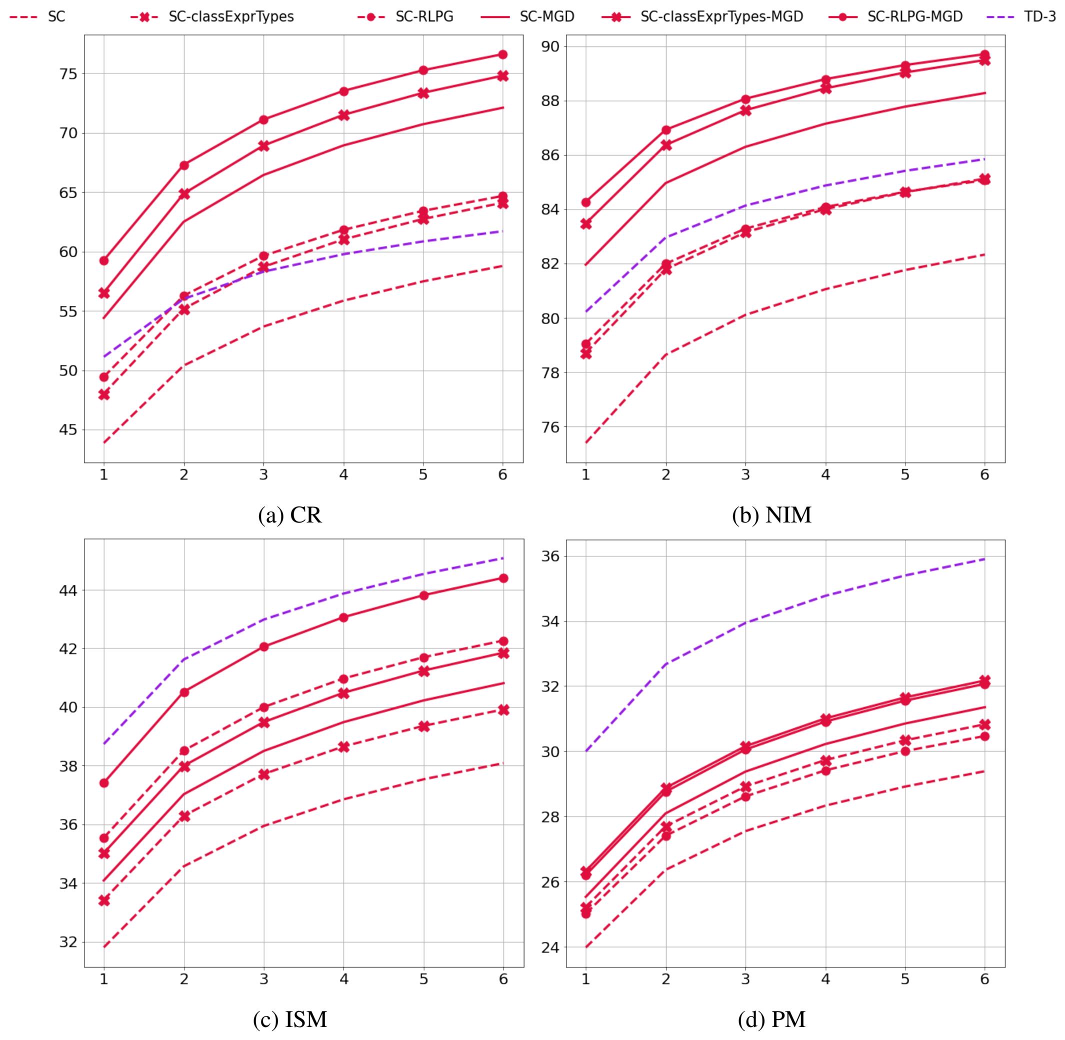
SantaCoder and CodeGen-2B with MGD outperform the larger CodeGen-6B with a relative margin of 10% and 4.9% respectively.

Prefix Match. Figure 2d shows percentage prefix match with ground truth. All the models with MGD outperform their respective base models with a relative improvement of 6.1%-6.7%. Both SantaCoder and CodeGen-2B with MGD outperform the larger CodeGen-6B.

SC-MGD vs. TD-3. SC is a 1.1B parameter model whereas TD-3 has 175B parameters. Being a much larger model and possibly due to other differences in training, TD-3 does better than SC across all metrics. Interestingly, with MGD, SC-MGD outperforms TD-3 on CR (Figure 2a) and NIM (Figure 2b). ISM and PM are sequence-level metrics and the relative advantage of the larger model prevails. Even then, with a small increase in the sampling budget, from $k = 1$ to $k = 4$, SC-MGD manages to surpass TD-3's performance with $k = 1$ on ISM (Figure 2c) and PM (Figure 2d).

Summary. MGD improves the compilability of code significantly, across architectures and parameter scale, leading even the smallest CodeGen-350M with MGD to outperform the largest LM, text-davinci-003. We also see improvements in all ground truth agreement metrics. Notably, smaller LMs with MGD outperform larger LMs (CodeGen-350M with MGD outperforms text-davinci-003 in CR and NIM, CodeGen-2B with MGD outperforms CodeGen-6B on ISM and PM) across all metrics.

4.2 Effect of MGD and Prompt Augmentation Strategies



(a) CR

(b) NIM

(c) ISM

(d) PM

Figure 3: score@k for models with MGD and prompt augmentation compared against base models

We choose the best-performing base model, SantaCoder, from study 4.1 to study the effect of prompting when combined with MGD. Figure 3 shows results for SantaCoder-{Standard, classExprTypes, RLPG} and text-davinci-003, compared against base models with respective prompts and MGD.

Compilation Rate. Figure 3a shows the results for Compilation Rate. We observe improvements in compilation rate with both the prompting techniques, classExprTypes and RLPG, with RLPG marginally outperforming classExprTypes. We note that SantaCoder with Standard prompt and MGD is able to relatively improve over both RLPG and classExprTypes augmentation by 11% and 12% respectively. Further, SantaCoder with RLPG and MGD is able to outperform SantaCoder-RLPG and SantaCoder-classExprTypes with a relative margin of 18% and 20% respectively, while increasing the margin of relative improvement over text-davinci-003 to 24%.

Next Identifier Match. As seen in 3b, similar to compilation, both RLPG and classExprTypes prompt augmentation lead to improvement over the base model. However, SantaCoder with either prompt augmentations underperforms text-davinci-003. SantaCoder with MGD outperforms SantaCoder-RLPG and SantaCoder-classExprTypes with a relative improvement of 3.8% and 3.7% respectively. SantaCoder with prompting and MGD outperform their respective baselines (SantaCoder with prompting) by a relative margin in the range of 5.1%-5.5%. We note that SantaCoder with RLPG and MGD increases the relative improvement with respect to the largest model, text-davinci-003 to 4.5%.

Identifier Sequence Match. On ISM, SantaCoder with prompt augmentation and MGD is able to outperform its respective baseline by a relative improvement of 4.8%-5.1%, while both the prompt augmentations result in an improvement over the base model. SantaCoder with RLPG and MGD is able to significantly reduce the gap with text-davinci-003, underperforming it by just 1.5%.

Prefix Match. As seen in Figure 3d, SantaCoder with prompt augmentation and MGD is able to outperform its respective baseline by 4.4%-5.3%.

Summary. While prompt augmentation techniques, classExprTypes and RLPG both help in improving performance on all metrics, we see further improvement with MGD augmentation, and conclude that the contributions by prompt augmentation and MGD are complementary. Notably, SantaCoder-RLPG with MGD improves the relative margin for compilation rate with respect to text-davinci-003 to 24% compared to the 17% improvement achieved by SC-MGD.

4.3 Effect of MGD on Fill-in-the-middle (FIM) Decoding

Among the base models, SantaCoder supports the FIM modality. We evaluated SantaCoder with autoregressive and FIM decoding strategies and text-davinci-003, and compared them with respective configurations of SantaCoder with MGD. Similar to our observations with prompt augmentation, while FIM modality leads to improvements across all metrics, we see continued improvement when using both FIM and MGD. Due to space limitations, detailed results are in Appendix E. Motivated by the complementary nature of FIM and MGD, we further evaluated SC-FIM-classExprTypes-MGD, combining both prompt augmentation and FIM modality. Consistent with our findings, it leads to a further improvement over SC-FIM-classExprTypes, as seen in Figure 5 (Appendix E).

4.4 Effect of Identifier Complexity on Next Identifier Match

Identifier names in code repositories can often get specific and long (Karampatsis et al., 2020). Due to this, while commonly used APIs may get tokenized into single tokens, identifiers specific to the context of individual repositories, especially in private settings, can span over multiple subtokens in the LM vocabulary. We define the complexity of an identifier as the mean number of subtokens required to decode it. We show that the ability of LMs to accurately predict the identifier name decreases sharply with an increase in identifier complexity, while augmenting them with MGD improves their performance. The detailed results are available in Appendix F.

5 Discussion

Limitations. Though static analysis is a mature field with strong theoretical foundations and several robust implementations of tools, analyzing partial and incomplete programs is still a difficult problem. In practice, editors such as Eclipse and Visual Studio support static analysis with heuristics. Though these heuristics are well-engineered and are widely used, they can be both imprecise (they can give incorrect suggestions) and incomplete (they can leave out correct suggestions). In addition, correctness of generations depends on the specification of intended functionality, which is beyond the scope of this work. Consequently, though our results from guiding LMs using these analyses (through MGD) show improvements in quality metrics, additional steps such as testing and human inspection are needed to guarantee correctness of generated code.

Societal Impact. Software pervasively affects all aspects of our lives. With LMs being widely deployed as copilots and intelligent assistants to help developers write code, it is crucially important to develop tools like MGD to improve the quality of code generated by LMs (even if humans review and accept the suggestions given by LMs). Without such tools, we risk introducing bugs in code due to incorrect suggestions made by LMs, which has the potential to impact all of our lives negatively.

Amount of Compute. Our experiments do not involve any training, and we only perform inferences. We used machines of the following configurations: (1) CPU: 24-core AMD Epyc with 220GB RAM, GPU: Nvidia A100 80GB. (2) CPU: Intel Xeon(R) Platinum 8168 with 290GB RAM, GPU: Nvidia Tesla V100 16GB. For the experiments to evaluate text-davinci-003, we used inference access to an instance of the model hosted in Azure.

6 Related Work

Pre-trained Models of Code. Many powerful pre-trained models have been designed for code. These include encoder-only models like CodeBERT (Feng et al., 2020), GraphCodeBERT Guo et al. (2020) and CuBERT (Kanade et al., 2020); encoder-decoder models like PLBART (Ahmad et al., 2021), CodeT5 (Wang et al., 2021) and AlphaCode (Li et al., 2022); or decoder-only models like Codex (Chen et al., 2021), GPT-J (Wang & Komatsuzaki, 2021), Austin et al. (2021), GPT-Neo (Black et al., 2021), GPT-NeoX (Black et al., 2022), CodeParrot (Tunstall et al., 2022), PolyCoder (Xu et al., 2022), Incoder (Fried et al., 2022), CodeGen (Nijkamp et al., 2023), SantaCoder (Allal et al., 2023) and StarCoder (Li et al., 2023); or unified models like UniXCoder (Guo et al., 2022). Note that these are representative examples and are not meant to be an exhaustive list. Our monitor-guided decoding works only with logits and hence can be used with any model.

Global Context of Code. The context required for understanding and generating code may extend well-beyond the code in the vicinity of the location of interest. Hellendoorn & Devanbu (2017) build an n-gram model with a cache to keep track of the directory-level context. Xu et al. (2021) use locality based on directory-structure in retrieval-augmented modeling (Khandelwal et al., 2019). Many approaches use static analysis or previous generations (Zhang et al., 2023) to extract relevant code entities. They use the relevant context to either augment the prompt (Shrivastava et al., 2022; Pei et al., 2023; Zhang et al., 2023) or embeddings (Pashakhanloo et al., 2022; Ding et al., 2022) presented as input to the LM. Due to the limit on the prompt or embedding size, these approaches filter information through random walks (Pashakhanloo et al., 2022), classification (Shrivastava et al., 2022) or using fixed pruning strategies (Ding et al., 2022).

As we neither augment the prompt nor use extra embeddings, we do not need to prune the global context. We let the static analysis generate completion suggestions using the entire repository-level context. Zan et al. (2022) and Zhou et al. (2022) retrieve information from library documentation for prompt augmentation. Our static analysis analyzes libraries along with the repository-level source code. Several of these techniques require architecture modifications (Pashakhanloo et al., 2022; Ding et al., 2022) or finetuning (Zan et al., 2022; Ding et al., 2022; Pei et al., 2023). We use a simple interface between logits and static analysis with a frozen LM. Most of these approaches, excluding (Xu et al., 2021; Zhang et al., 2023), use one-time *a priori* retrieval. In contrast, we provide token-level guidance by invoking static analysis on demand. Our method is *complementary* to all the above approaches as they all try to condition the generation by modifying the *input* to the LM whereas we apply *output*-side constraints by reshaping the logits. Indeed, our experiments in Section 4.2 show that combining our method with prompt-augmentation gives additional boost in performance.

Syntactic and Semantic Constraints. There are two primary lines of work to enforce syntactic and semantic constraints on code generation, based on specialized modeling and through constrained decoding. GNN2NAG (Brockschmidt et al., 2019) and NSG (Mukherjee et al., 2021) are examples of the first and use attribute grammars. GNN2NAG computes attributes neurally whereas NSG does it symbolically, and are respectively evaluated on expressions that do not use user-defined methods or on methods with class-level context. We consider the complete repository-level context for method-level completion. Unlike these approaches, our objective is to guide pre-trained LMs without architecture modification or additional training. PICARD (Scholak et al., 2021) and Synchromesh (Poesia et al., 2022) fall in the category of constrained decoding similar to ours. These approaches use incremental parsing for syntactic validity and design domain-specific semantic checks to ensure semantic validity, e.g., an SQL expression uses only columns defined for a table. Both are evaluated on SQL, and Synchromesh additionally considers domain-specific languages for visualization and calendar applications. In comparison, we target general-purpose programming languages with focus on type-consistent identifier generation using static analysis over repository-level context.

7 Conclusions and Future Work

Using static analysis tools from IDEs, designed to aid developers, to help LMs generate code with improved quality is under-explored. In this work, we show how to use repository-wide information computed by static analysis (specifically, type-based analysis) using a stateful monitor as an interface, to improve quality of code generated by LMs. Our experimental results show the potential for significant quality improvements for Java code generation using this approach. Our approach is complementary to prompt augmentation techniques. It allows smaller models to achieve better or competitive performance compared to much larger models. This could open up the possibility of using such smaller models directly within IDEs, alongside our monitor, as an alternative method to the use of remotely-hosted Large LMs (LLMs), reducing inference costs and improving privacy.

In future work, we plan to explore the efficacy of MGD for code written in other widely used programming languages. In addition, we plan to expand the scope of static analysis to deeper semantic analysis such as type-state analysis (Strom & Yemini, 1986; Fink et al., 2008) and joint monitoring based on multiple static analyses.

References

Project Lombok. `https://projectlombok.org/`. Accessed: May 14, 2023.

Official page for Language Server Protocol. URL `https://microsoft.github.io/language-server-protocol/`.

Protocol Buffers. `https://protobuf.dev/`. Accessed: May 14, 2023.

Eclipse JDT LS, September 2016. URL `https://projects.eclipse.org/projects/eclipse.jdt.ls`.

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation, 2021.

Allal, L. B., Li, R., Kocetkov, D., Mou, C., Akiki, C., Ferrandis, C. M., Muennighoff, N., Mishra, M., Gu, A., Dey, M., and others. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program Synthesis with Large Language Models, August 2021. URL `http://arxiv.org/abs/2108.07732`. arXiv:2108.07732 [cs].

Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., and Chen, M. Efficient training of language models to fill in the middle, 2022.

Black, S., Gao, L., Wang, P., Leahy, C., and Biderman, S. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58, 2021.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., and others. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative Code Modeling with Graphs. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=Bke4KsA5FX`.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., and others. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Dagenais, B. and Hendren, L. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pp. 313–328, 2008.

Ding, Y., Wang, Z., Ahmad, W. U., Ramanathan, M. K., Nallapati, R., Bhatia, P., Roth, D., and Xiang, B. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context, December 2022. URL `http://arxiv.org/abs/2212.10007`. arXiv:2212.10007 [cs].

Donahue, C., Lee, M., and Liang, P. Enabling language models to fill in the blanks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2492–2501, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.225. URL `https://aclanthology.org/2020.acl-main.225`.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and others. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Fink, S. J., Yahav, E., Dor, N., Ramalingam, G., and Geay, E. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), may 2008. ISSN 1049-331X. doi: 10.1145/1348250.1348255. URL `https://doi.org/10.1145/1348250.1348255`.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Fuhrer, R. M. Leveraging static analysis in an ide. *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, pp. 101–158, 2013.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., and others. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. Unixcoder: Unified cross-modal pre-training for code representation, 2022.

Hedin, G. Incremental semantic analysis. *Department of Computer Sciences*, 1992.

Hellendoorn, V. J. and Devanbu, P. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=rygGQyrFvH`.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pp. 5110–5121. PMLR, 2020.

Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., and Janes, A. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1073–1085, 2020.

Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*, 2019.

Kudo, T. and Richardson, J. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., Freitas, N. d., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL `https://www.science.org/doi/abs/10.1126/science.abq1158`. _eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158.

Maddox III, W. H. *Incremental static semantic analysis*. University of California, Berkeley, 1997.

Mukherjee, R., Wen, Y., Chaudhari, D., Reps, T., Chaudhuri, S., and Jermaine, C. Neural Program Generation Modulo Static Analysis. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, 2021. URL `https://openreview.net/forum?id=yaksQCYcRs`.

Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer, 2015.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=iaYcJKpY2B_`.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback, 2022.

Pashakhanloo, P., Naik, A., Wang, Y., Dai, H., Maniatis, P., and Naik, M. Codetrek: Flexible modeling of code using an extensible relational representation. 2022.

Pei, H., Zhao, J., Lausen, L., Zha, S., and Karypis, G. Better context makes better code language models: A case study on function call argument completion. In *AAAI*, 2023. URL `https://www.amazon.science/publications/better-context-makes-better-code-language-models-a-case-study-on-function-call-argument-completion`.

Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. Synchromesh: Reliable code generation from pre-trained language models. In *ICLR 2022*, April 2022. URL `https://www.microsoft.com/en-us/research/publication/synchromesh-reliable-code-generation-from-pre-trained-language-models/`.

Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):449–477, 1983.

Scholak, T., Schucher, N., and Bahdanau, D. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL `https://aclanthology.org/2021.emnlp-main.779`.

Schuster, M. and Nakajima, K. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp. 5149–5152. IEEE, 2012.

Sennrich, R., Haddow, B., and Birch, A. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL `https://aclanthology.org/P16-1162`.

Shrivastava, D., Larochelle, H., and Tarlow, D. Repository-level prompt generation for large language models of code. *arXiv preprint arXiv:2206.12839*, 2022.

Strom, R. E. and Yemini, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1):157–171, 1986.

Tunstall, L., Von Werra, L., and Wolf, T. *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

Wagner, T. A. *Practical algorithms for incremental software development environments*. University of California, Berkeley, 1997.

Wang, B. and Komatsuzaki, A. GPT-J-6B: A 6 billion parameter autoregressive language model, 2021.

Wang, Y., Wang, W., Joty, S. R., and Hoi, S. C. H. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv*, abs/2109.00859, 2021.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL `https://aclanthology.org/2020.emnlp-demos.6`.

Xu, F. F., He, J., Neubig, G., and Hellendoorn, V. J. Capturing structural locality in non-parametric language models. *arXiv preprint arXiv:2110.02870*, 2021.

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, pp. 1–10, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9273-0. doi: 10.1145/3520312.3534862. URL `https://doi.org/10.1145/3520312.3534862`. event-place: San Diego, CA, USA.

Yu, H., Shen, B., Ran, D., Zhang, J., Zhang, Q., Ma, Y., Liang, G., Li, Y., Xie, T., and Wang, Q. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models, February 2023. URL `http://arxiv.org/abs/2302.00288`. arXiv:2302.00288 [cs].

Zan, D., Chen, B., Lin, Z., Guan, B., Yongji, W., and Lou, J.-G. When Language Model Meets Private Library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 277–288, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL `https://aclanthology.org/2022.findings-emnlp.21`.

Zhang, F., Chen, B., Zhang, Y., Liu, J., Zan, D., Mao, Y., Lou, J.-G., and Chen, W. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. *arXiv preprint arXiv:2303.12570*, 2023.

Zhou, S., Alon, U., Xu, F. F., Jiang, Z., and Neubig, G. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2022.

Supplementary Material for: Guiding Language Models of Code with Global Context using Monitors

Outline

- Monitor for the Running Example
- Data Set Curation Details
- Experimental Setup - Additional Details
- Calculation of score@k
- Effect of MGD on Fill-in-the-middle (FIM) Decoding - Complete Results
- Effect of Identifier Complexity on Next Identifier Match - Complete Results
- Impact of MGD on Inference Time
- CodeQL Query for Identifying Evaluation Target Methods
- Examples of code generation with MGD

A Monitor for the Running Example

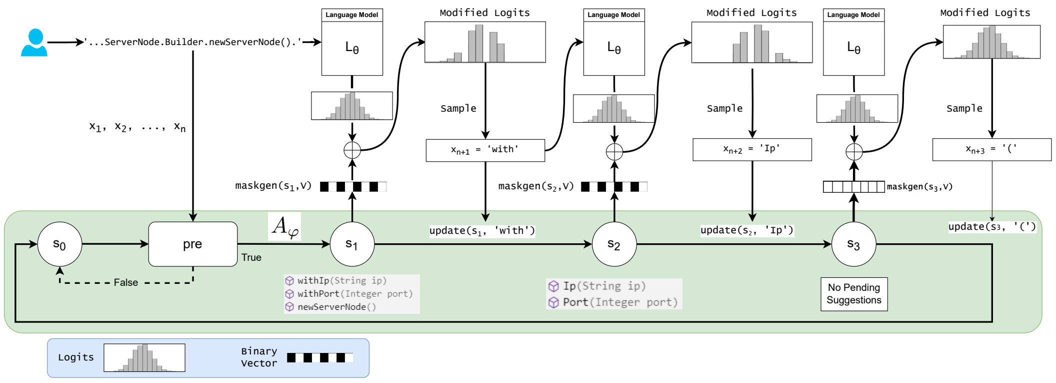


Figure 4: Monitor to guide generating type-consistent identifers for the code in Figure 1.

Figure 4 shows how the monitor interacts with the LM decoder for the example in Figure 1. Initially, the monitor, M_φ is in the wait state s_0. Given the code completion input, $x_1, x_2, ..., x_n$, M_φ first evaluates $pre(s_0, x_1, ..., x_n)$. Since x_n = '.' (the dot symbol indicating object dereferencing in Java), $pre(s_0, x_1, ..., x_n)$ evaluates to $true$, and subsequently, in accordance with Eq. (4), the static analysis A_φ is invoked, which determines the input prompt to be in accordance with the property φ, and resolves the type for the completion point to be `ServerNode.Builder`, as shown in the annotated AST in Figure 1(b). A_φ then returns the set of identifiers consistent with the resolved type – {`withIp, withPort, newServerNode, ...`}, transitioning the monitor M_φ to state s_1. M_φ then calculates $m = maskgen(s_1, V)$, which masks out, for example, the token `host` (as inferred by SantaCoder in Figure 1(b)). Concurrently, the input is tokenized and the LM L_θ provides inferred logits, ℓ for the next token. The output logits ℓ from L_θ and mask m are combined as $\ell \oplus m$ to obtain the modified logits, which is then softmaxed and a token is sampled –`with` in this case. The monitor then invokes $update(s_1, \texttt{with})$ to transition to s_2. Note that with the state transition, `newServerNode` is pruned from the set of identifiers, as the sampled token `with` does not prefix it. L_θ provides logits for the next token, and the composition of L_θ and M_φ is repeated to obtain modified logits, and finally we obtain the sampled token `Ip`. Since `Ip` is a prefix of a member in s_2, $update(s_2, \texttt{Ip})$ transitions to

s_3, with $s_3 = \{\epsilon\}$, a singleton consisting of the empty string. Note that the token '`(`' is consistent with the suggestions in state s_3, since it matches the regular expression $w \cdot E \cdot \Sigma^*$, where $w = \epsilon$. Following the sampling of the token '`(`', the monitor transitions back to the wait state s_0.

B Data Set Curation Details

For the evaluation of pragmatic code generation, we require real-world repositories, with their complete environments and dependencies. Most public datasets for code generation do not meet this requirement, due to their focus on standalone code generation, except the recent CoderEval (Yu et al., 2023) and PyEnvs (Pei et al., 2023), both of which are not publicly available at the time of this writing. Further, CoderEval just evaluates 10 Java and 43 Python repositories. Since these datasets do not filter for repositories by their creation date, the test repositories could be a part of the training set for the evaluated LMs. Considering these, we describe the curation of PRAGMATICCODE, a dataset of real-world open-source Java projects complete with their development environments and dependencies. We ensure that the training repositories were released publicly only after the determined training dataset cutoff date (31 March 2022) for the CodeGen (Nijkamp et al., 2023), SantaCoder (Allal et al., 2023), and text-davinci-003 (GPT-3.5) (Ouyang et al., 2022) family of models. All the repositories in PRAGMATICCODE are buildable with their respective build systems. The development environment also provides support for analysis over templated code generated with systems like ProtoBuf (pro) and Lombok (lom). Further, we create DOTPROMPTS from PRAGMATICCODE for the evaluation of code generation in a pragmatic setting.

PRAGMATICCODE We queried the GitHub API on 25 March 2023 to obtain a list of the top 1000-starred Java repositories on GitHub created after the determined cutoff date of 31 March 2022. We then attempt to download the latest snapshot of each of the top 1000-starred Java repositories and were able to download 731 repositories from GitHub. We create a build environment for Java projects consisting of Oracle Java Development Kit 17.0.6, Apache Ant 1.10.13, Apache Maven 3.8.7, Gradle 7.3.3, and GitHub CodeQL CLI 2.12.5. In this build environment, we invoke the CodeQL database create command [2] for every software repository obtained from GitHub. The CodeQL database creation process identifies the build system used in the repository and invokes the command for a clean build. The respective build systems use the project-level dependency information stored in configuration files like *pom.xml* and *build.gradle* to fetch the dependencies and store them locally. We filter out the repositories for which the CodeQL database creation failed, as that indicates that the repository either uses an unrecognized build system, some of its dependencies aren't satisfied, or the repository is in a transient state. We are left with 302 GitHub repositories after filtering for successful builds. We store the CodeQL database created for each of these repositories. Finally, we invoke the initialization of Eclipse JDT.LS[3] on each of the repositories, and filter for the repositories where JDT.LS could be successfully initialized. The final filtered list of 151 repositories, along with their CodeQL databases, comprise the PRAGMATICCODE dataset.

DOTPROMPTS. Yu et al. (2023) show that standalone functions account for less than 30% of open source projects among the top 100 most popular open source projects on GitHub, with most functions referencing third-party APIs or variables/constants defined in cross-file context. Hence, we create DOTPROMPTS for the evaluation of code generation in a pragmatic setting and aim to evaluate over real-world projects, not restricting to standalone function generation. Each test case in DOTPROMPTS consists of a prompt up to a dereference location (dereference '.' operator in Java) within a target method, and the task is to complete the remainder of the method. Since dereference locations are the points of occurrence for cross-file entities in source code, a model's ability to use cross-file context can be evaluated using DOTPROMPTS. **Curation Details.** We identify non-test class files that aren't auto-generated and have at least 1 cross-file dependency in the repository. From these files, we identify methods that aren't class and object initializers and have ≥ 2 top-level statements spanning ≥ 7 lines of source code, to ensure sufficient complexity in target methods. We use the CodeQL Query listed in section H to identify target methods from each of the repositories in PRAGMATICCODE based on the above criteria. As shown by Shrivastava et al. (2022), repositories are quite uneven in terms of their size, so to avoid individual repositories from dominating our evaluation, we limit to including up to 20 methods from each repository as identified by the CodeQL query. In order to simulate the real-world usage scenario, where a developer may invoke code completion at different

[2] https://docs.github.com/en/code-security/codeql-cli/using-the-codeql-cli/creating-codeql-databases

[3] https://github.com/eclipse/eclipse.jdt.ls

points within a method, we identify up to 10 uniformly distributed dereference locations within each of the identified methods. Each such dereference location becomes a data point for DOTPROMPTS.

C Experimental Setup - Additional Details

Monitor Implementation. Language Server Protocol (LSP) (lsp), is an open industry standard of communication between IDEs and programming language specific tools like static analyzers and compilers, called language servers. Eclipse JDT.LS (ecl, 2016) is a language server for Java, providing access to results of various static analyses over LSP. We instantiate Eclipse JDT.LS as an engine that implements the static analysis A_φ to check for type-consistency of identifiers. We implement our monitor M_φ as a thin layer around an LM, in accordance with Section 2, as a language server client that communicates with Eclipse JDT.LS. Since our implementation is based on LSP, and LSP is compatible with most major programming languages, our implementation can be easily ported to other languages besides Java.

Hyperparameters. We use nucleus sampling (Holtzman et al., 2020) with a top-p value of 0.95 to generate 6 samples in total—1 each with temperature 0.2 and 0.4, and 2 each with temperature 0.6 and 0.8. We fix a prompt budget of (2048-512)=1536 tokens, and generation budget of 512 tokens, for a total context window size of 2048. If the text exceeds the prompt budget, we truncate it from the left for standard and classExprTypes prompts, and from the right for FIM. For classExprTypes augmentation with autoregressive decoding, we reserve a budget of 20% of total prompting budget for classExprTypes, and the remaining for autoregressive prompt. For FIM decoding without prompt augmentation, we reserve 50% of total prompting budget for the suffix. For FIM decoding with classExprTypes prompt augmentation, we reserve 20% of total prompting budget for classExprTypes, 40% for suffix, and the remaining for autoregressive prompt.

D Calculation of score@k

Let $S = \{s_1, s_2, ..., s_n\}$ be a multiset representing metric scores obtained across n-independent trials. Without loss of generality, we order S in monotonic decreasing order as $S_{\geq} = (s_1^{\geq}, s_2^{\geq}, ..., s_n^{\geq})$. We calculate $score@k, n$ as per the equation below:

$$score@k, n = \frac{1}{\binom{n}{k}} \sum_{i=1}^{n-k+1} \binom{n-i}{k-1} S_{\geq}[i] = \frac{1}{\binom{n}{k}} \sum_{T \in \binom{S}{k}} \max(T) \tag{5}$$

$$\binom{S}{k} = \{V | V \subseteq S, |V| = k\} \tag{6}$$

where $1 \leq k \leq n$, and $\binom{S}{k}$ is the set of all subsets of S having cardinality k.

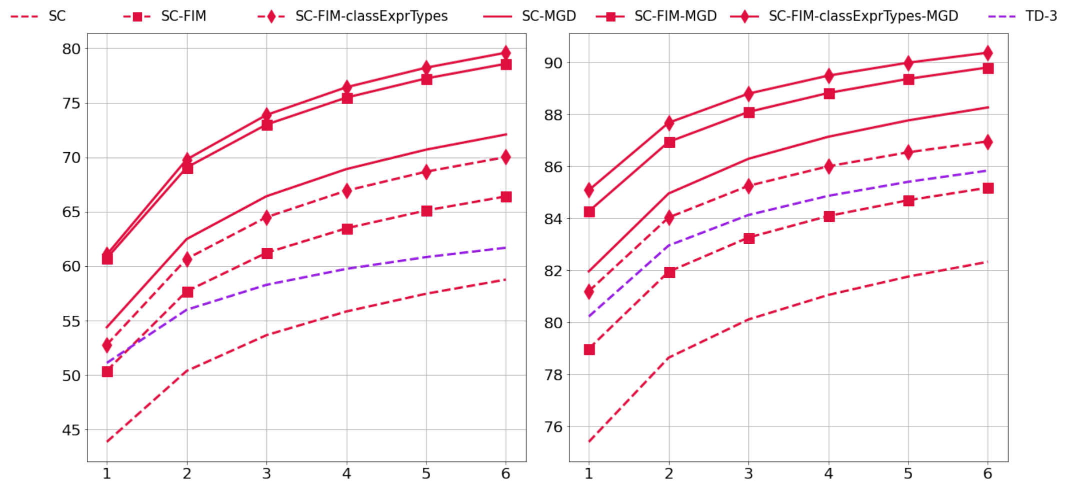
E Effect of MGD on Fill-in-the-middle (FIM) Decoding - Complete Results

Among the base models, SantaCoder supports the FIM modality. Figure 5 shows the results for SantaCoder with autoregressive and FIM decoding strategies and text-davinci-003, compared with respective configurations of SantaCoder with MGD.

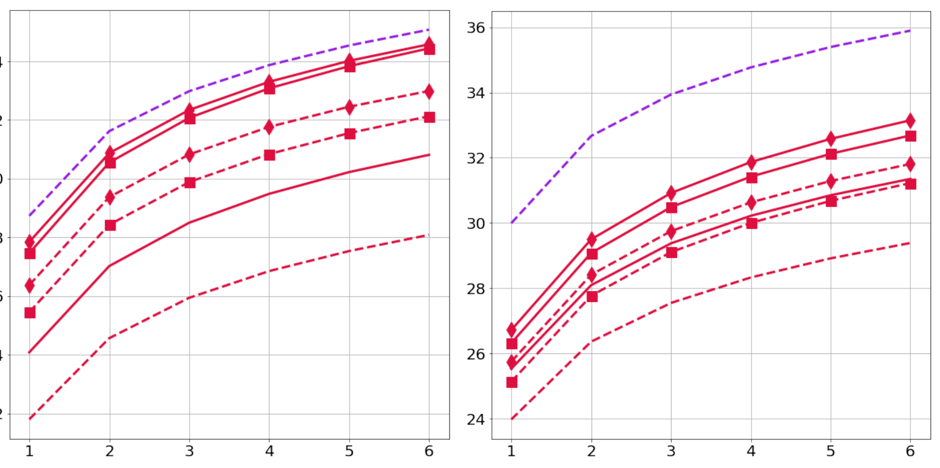
Compilation Rate. Figure 5a shows that SantaCoder with MGD outperforms SantaCoder-FIM by a relative margin of 8.6%. We see significant improvement in compilation rate, when SantaCoder-FIM is augmented with MGD, leading it to outperform text-davinci-003 with a relative margin of 27%. SantaCoder-FIM with MGD relatively improves over SantaCoder-FIM by 18%.

Next Identifier Match. Figure 5b shows that FIM modality boosts next identifier match but still underperforms text-davinci-003 and correspondingly, SantaCoder with MGD. Augmenting SantaCoder-FIM with MGD leads to a relative improvement of 5.4%.

Identifier Sequence Match. On ISM, SantaCoder-FIM with MGD improves over SantaCoder-FIM by 5.5%, which closes the gap with text-davinci-003, underperforming it by just 1.4%.



(a) CR (b) NIM



(c) ISM (d) PM

Figure 5: score@k for models with MGD and FIM compared against base models

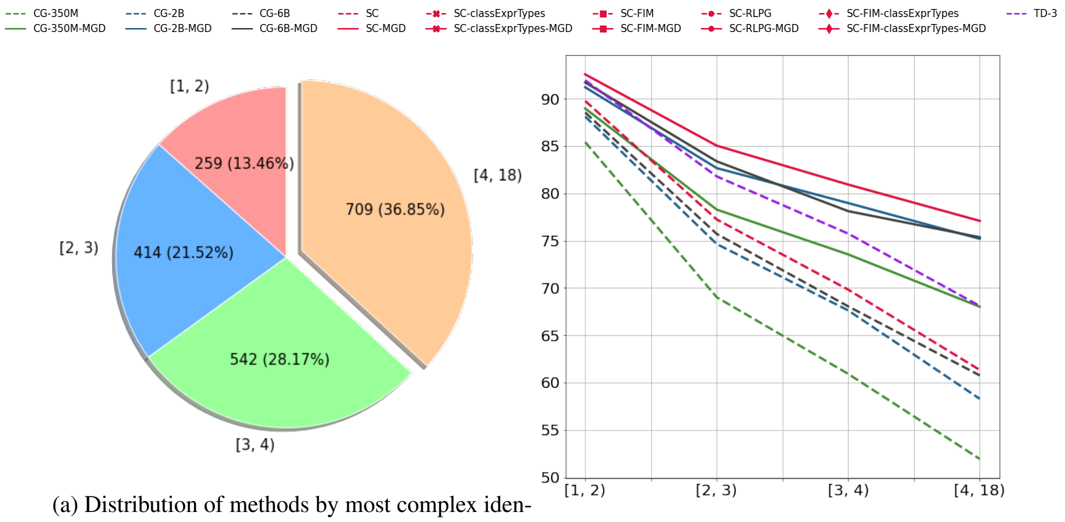
Prefix Match. Figure 5d shows that SantaCoder-FIM improves over SantaCoder, but still underperforms SantaCoder with MGD. SantaCoder-FIM with MGD outperforms SantaCoder-FIM by 4.7% showing continued improvements.

Summary. Similar to our observations with prompt augmentation, while FIM modality leads to improvements across all metrics, we see continued improvement when using both FIM and MGD.

SC-FIM-classExprTypes-MGD. Motivated by the complementary nature of MGD, we further evaluated SC-FIM-classExprTypes-MGD, combining both prompt augmentation and FIM modality, and consistent with our findings, it leads to a further improvement over SC-FIM-classExprTypes, as seen in Figure 5. We use classExprTypes with FIM since RLPG also selects a subset of post-lines in prompt augmentation in a large number of cases.

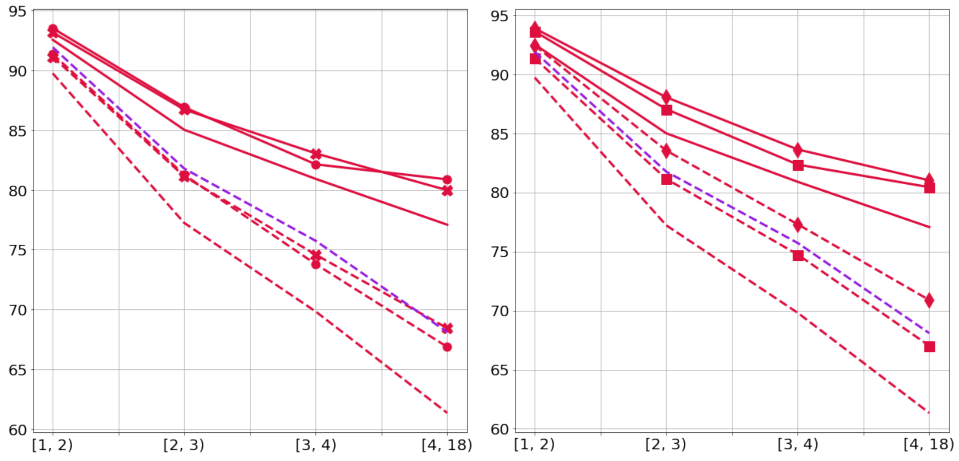
F Effect of Identifier Complexity on Next Identifier Match - Complete Results

Identifier Complexity. Identifier names in code repositories can often get specific and long (Karampatsis et al., 2020). The vocabulary of LMs like CodeGen and SantaCoder is generally created by training a BPE tokenizer over the complete or a sample of the pretraining dataset (Sennrich et al., 2016). Due to this, while commonly used APIs may get tokenized into single tokens, identifiers specific to the context of individual repositories, especially in private settings, can span over multiple subtokens in the LM vocabulary, making their accurate generation difficult for the LM (both due to



(a) Distribution of methods by most complex identifier

(b) Standard prompt



(c) Prompt augmentation

(d) FIM

Figure 6: (NIM, score@6) across next-identifier complexity

the increased number of decoding steps leading to a larger search space and also due to the relative rarity of the identifier name). We define the complexity of an identifier as the mean number of subtokens required to decode it, using the tokenizers of all the models under study (CG, SC, TD-3). About 36.85% of methods in DOTPROMPTS dataset have at least 1 identifier of complexity [4, 18) as shown in Figure 6a, and therefore, for a model to generate those methods correctly, it is important for the model to be able to generate the complex identifier name.

Next Identifier Match. Figure 6 shows the result for the NIM metric, across increasing complexity of the next ground-truth identifier. We note that all models show a sharp degradation in performance with an increase in identifier complexity. The same trend holds true for prompt augmentation as well as FIM modality. Augmenting base models with MGD leads to significant improvement in the model's ability to predict the next identifier for the most complex identifier case [4, 18) with a relative improvement in the range of 24%-31%. The smallest model, CodeGen-350M with MGD achieves parity with the largest model, text-davinci-003 and outperforms the much larger CodeGen-6B with a relative margin of 12%. CodeGen-2B with MGD outperforms CodeGen-6B and text-davinci-003 by a relative margin of 24% and 10% respectively. SantaCoder with MGD improves over text-davinci-003 by a relative margin of 13%. We further observe that both prompt augmentation and FIM-modality with MGD lead to a diminishing rate of degradation, as can be seen in the curves for SC-FIM-MGD in Figure 6d. SantaCoder-FIM with MGD outperforms text-davinci-003 and SantaCoder-FIM by a relative margin of 18% and 20% respectively.

Table 2: Statistics on inference time comparing CodeGen-6B and Codegen-6B with MGD. Time in seconds.

	CodeGen-6B	CodeGen-6B-MGD
Mean	22.57	41.34
Mean slowdown		83.16%
Standard deviation	12.44	22.39
Min	0.94	1.14
First quartile	21.23	26.76
Median	25.48	44.56
Second quartile	25.86	58.20
Max	79.06	111.73

Summary. The ability of LMs to accurately predict the identifier name decreases sharply with an increase in identifier complexity. All models with MGD outperform their respective base models with a large relative improvement in the range of 24%-31%, with smaller LMs outperforming much larger LMs (CodeGen-350M-MGD achieves parity with text-davinci-003, and outperforming CodeGen-6B. CodeGen-2B-MGD outperforms CodeGen-6B). While prompt augmentation and FIM modality lead to improvement over baselines, they also suffer from a sharp decrease across identifier complexity, but augmenting them with MGD leads to similar large improvements as observed with base models (relative improvement in the range of 17%-21%).

G Impact of MGD on Inference Time

To study the impact of adding MGD on inference time, we compare the code generation times by CodeGen-6B model and CodeGen-6B with MGD. We select 500 prompts from DotPrompts and generate up to 512 tokens with CodeGen-6B as well as CodeGen-6B with MGD. The inferencing is performed with HuggingFace implementation of the model on machine configuration (1) as described in Section 5. We use the same decoding scheme as described in Section 3. After generation, we filter out the cases where the number of tokens generated by CodeGen-6B and Codegen-6B with MGD is not the same, in order to ensure parity both in the size of the input prompt and number of generated tokens. We are left with 161 instances, where the size of the input prompt, as well as the number of generated tokens, is the same for both models. Figure 2 shows statistics related to inference time for both the models in seconds. We observe a mean slowdown of 83.16% in decoding time for CodeGen-6B with MGD. We note that there might be many opportunities to optimize our implementation though we have not explored them yet.

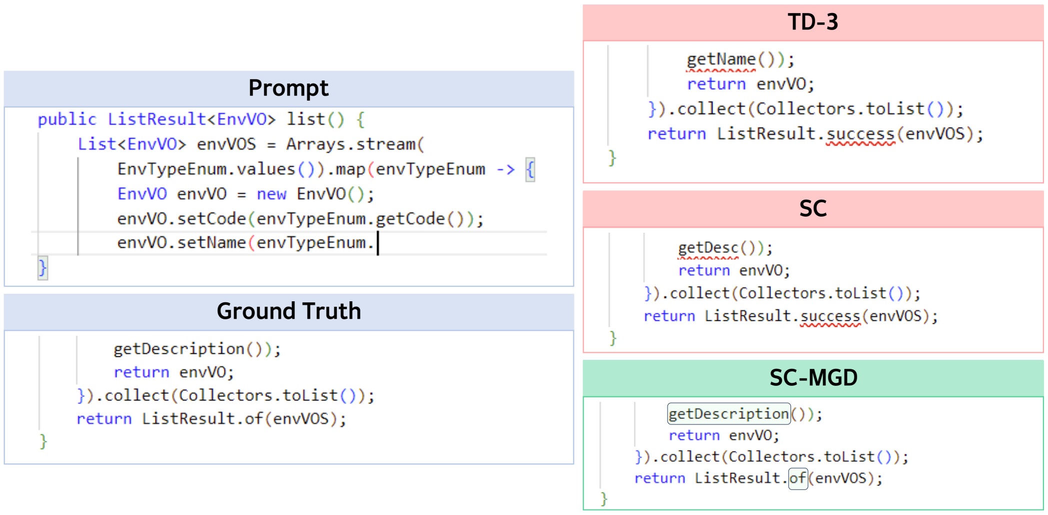
H CodeQL Query for Identifying Evaluation Target Methods

```
/**
 * @id java/examples/find_target_methods
 * @name find_target_methods
 * @description Identify target methods from a Java repository along with
     classExprTypes information for DotPrompts dataset
 */

import java

predicate filterClass(Class c) {
       not(c instanceof TestClass) and
       c.fromSource() and
       not c.getFile().getRelativePath().matches("%generated%") and
       not(c.getFile().getRelativePath().matches("%test%")) and
       not(c.getFile().getRelativePath().matches("%target%")) and
       not(c.getFile().getRelativePath().matches("%build%")) and
       count(Method m1 | m1 = c.getAMethod() and filterMethod(m1) | m1) >= 1
}
```

```
predicate filterMethod(Method m) {
        m.fromSource() and
        not(m instanceof TestMethod) and
        not(m.hasName("<clinit>")) and
        not(m.hasName("<obinit>")) and
        m.getBody().getNumStmt() >= 2 and
        (m.getBody().getLocation().getEndLine() -
            m.getBody().getLocation().getStartLine()) >= 7
}

predicate typeDeclaredInFile(Type t, File file) {
        (t.fromSource() and t.getFile() = file) or
        (t.getErasure().fromSource() and t.getErasure().getFile() = file)
}

// Find classExprTypes files such that they contain type definition of any
    expressions defined in any callable in the class except for target_m
predicate filterClassExprTypeFile(Class c, Method target_m, File classFile, File
    classExprTypesFile){
        classExprTypesFile != classFile and
        (
                // classExprTypesFile contains type definition of a singly imported
                    type
                exists(Type t, ImportType impt |
                        impt.fromSource() and
                        c.getFile() = impt.getFile() and
                        t = impt.getImportedType() and
                        typeDeclaredInFile(t, classExprTypesFile)
                ) or

                // classExprTypesFile contains type definition of return type or
                    param type of the target method
                exists(Type t |
                        (t = target_m.getAParamType() or t =
                            target_m.getReturnType()) and
                        typeDeclaredInFile(t, classExprTypesFile)
                ) or

                // classExprTypesFile contains type definition of type of any
                    expression within a callable (that is not the target method) in
                    class c, or a return type of a callable or any of its parameters
                exists(Expr e, Type t, Callable m |
                        m = c.getACallable() and
                        m != target_m and
                        (
                                (
                                        e.getAnEnclosingStmt() = m.getBody().getAStmt()
                                            and
                                        e.getType() = t
                                ) or
                                t = m.getAParamType() or t = m.getReturnType()
                        ) and
                        typeDeclaredInFile(t, classExprTypesFile)
                ) or

                // classExprTypesFile contains type definition of any field type
                exists(Type t, Field f |
                        c.getAField() = f and
                        f.getType() = t and
                        typeDeclaredInFile(t, classExprTypesFile)
                )
        )
}

predicate expressionOfTypeContainedInBlock(Expr e, Type t, BlockStmt b) {
        e.getAnEnclosingStmt() = b.getAStmt() and
        e.getType() = t
}

from File classFile, File classExprTypesFile, Class c, Method m, BlockStmt b,
        int startLine, int startCol, int endLine, int endCol

where
        // Bind variables
        m = c.getAMethod() and
        b = m.getBody() and
        classFile = c.getFile() and

        // Apply filters
        filterClass(c) and
        filterMethod(m) and
        filterClassExprTypeFile(c, m, classFile, classExprTypesFile) and

        // Bind method boundary locations
        startLine = b.getLocation().getStartLine() and
        startCol = b.getLocation().getStartColumn() and
        endLine = b.getLocation().getEndLine() and
        endCol = b.getLocation().getEndColumn()

select
        classFile.getAbsolutePath(),
        classFile.getRelativePath(),
        classExprTypesFile.getAbsolutePath(),
        classExprTypesFile.getRelativePath(),
        startLine,
        startCol,
        endLine,
        endCol
```

I Examples of code generation with MGD

We present examples of code generation with MGD and compare them to generations without MGD augmentation. The appearance of red-markers below identifier names indicate that the identifier name is not type-consistent with the target object/class for the dereference operator. We highlight the correct identifiers generated with MGD augmentation in light green.



Prompt

```
public ListResult<EnvVO> list() {
    List<EnvVO> envVOS = Arrays.stream(
        EnvTypeEnum.values()).map(envTypeEnum -> {
        EnvVO envVO = new EnvVO();
        envVO.setCode(envTypeEnum.getCode());
        envVO.setName(envTypeEnum.
}
```

Ground Truth

```
        getDescription());
        return envVO;
    }).collect(Collectors.toList());
    return ListResult.of(envVOS);
}
```

TD-3

```
        getName());
        return envVO;
    }).collect(Collectors.toList());
    return ListResult.success(envVOS);
}
```

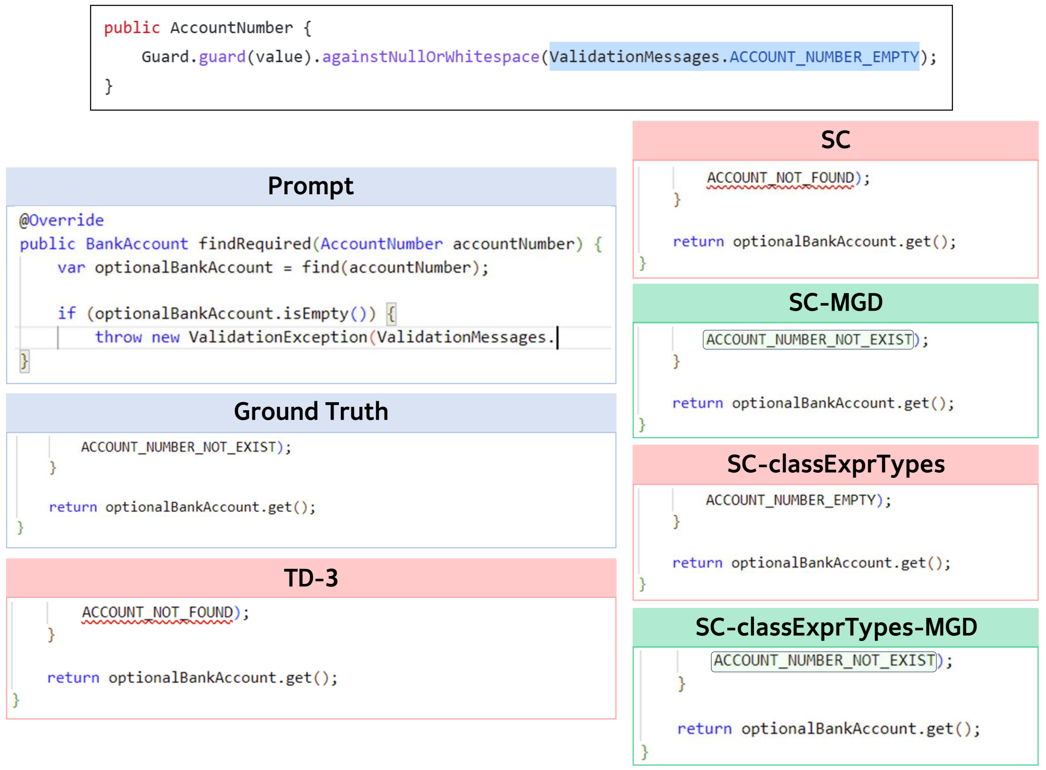
SC

```
        getDesc());
        return envVO;
    }).collect(Collectors.toList());
    return ListResult.success(envVOS);
}
```

SC-MGD

```
        getDescription());
        return envVO;
    }).collect(Collectors.toList());
    return ListResult.of(envVOS);
}
```

Figure 7: TD-3 and SC generate invalid identifier names: `getName`, `getDesc` for target object `envTypeEnum` and `success` for target class `ListResult`. Augmenting SC with MGD leads to the generation of correct identifiers: `getDescription`, `of` for the respective objects, leading to complete agreement with the ground truth.



```
public AccountNumber {
    Guard.guard(value).againstNullOrWhitespace(ValidationMessages.ACCOUNT_NUMBER_EMPTY);
}
```

Prompt

```
@Override
public BankAccount findRequired(AccountNumber accountNumber) {
    var optionalBankAccount = find(accountNumber);

    if (optionalBankAccount.isEmpty()) {
        throw new ValidationException(ValidationMessages.
}
```

Ground Truth

```
        ACCOUNT_NUMBER_NOT_EXIST);
    }

    return optionalBankAccount.get();
}
```

TD-3

```
        ACCOUNT_NOT_FOUND);
    }

    return optionalBankAccount.get();
}
```

SC

```
        ACCOUNT_NOT_FOUND);
    }

    return optionalBankAccount.get();
}
```

SC-MGD

```
        ACCOUNT_NUMBER_NOT_EXIST);
    }

    return optionalBankAccount.get();
}
```

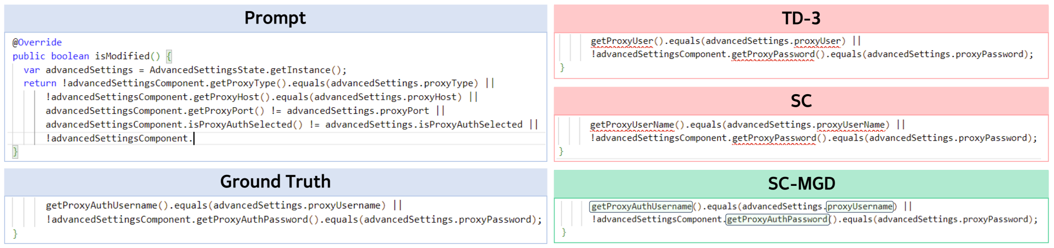
SC-classExprTypes

```
        ACCOUNT_NUMBER_EMPTY);
    }

    return optionalBankAccount.get();
}
```

SC-classExprTypes-MGD

```
        ACCOUNT_NUMBER_NOT_EXIST);
    }

    return optionalBankAccount.get();
}
```

Figure 8: SC and TD-3 generate the identifier name: `ACCOUNT_NOT_FOUND`, which is invalid for the target class. The snippet of code presented above is from one of the files present in the classExprTypes augmentation. Augmenting SC with classExprTypes leads the model to generate the identifier name `ACCOUNT_NUMBER_EMPTY`, which is consistent with the type: `ValidationMessages`, and therefore compilable. However, it still does not match the ground truth. Augmenting both SC and SC-classExprTypes with MGD leads the model to generate the correct identifier name: `ACCOUNT_NUMBER_NOT_EXIST`, achieving agreement with the ground truth.



Prompt

```java
@Override
public boolean isModified() {
  var advancedSettings = AdvancedSettingsState.getInstance();
  return !advancedSettingsComponent.getProxyType().equals(advancedSettings.proxyType) ||
      !advancedSettingsComponent.getProxyHost().equals(advancedSettings.proxyHost) ||
      advancedSettingsComponent.getProxyPort() != advancedSettings.proxyPort ||
      advancedSettingsComponent.isProxyAuthSelected() != advancedSettings.isProxyAuthSelected ||
      !advancedSettingsComponent.
}
```

Ground Truth

```java
      getProxyAuthUsername().equals(advancedSettings.proxyUsername) ||
      !advancedSettingsComponent.getProxyAuthPassword().equals(advancedSettings.proxyPassword);
}
```

TD-3

```java
      getProxyUser().equals(advancedSettings.proxyUser) ||
      !advancedSettingsComponent.getProxyPassword().equals(advancedSettings.proxyPassword);
}
```

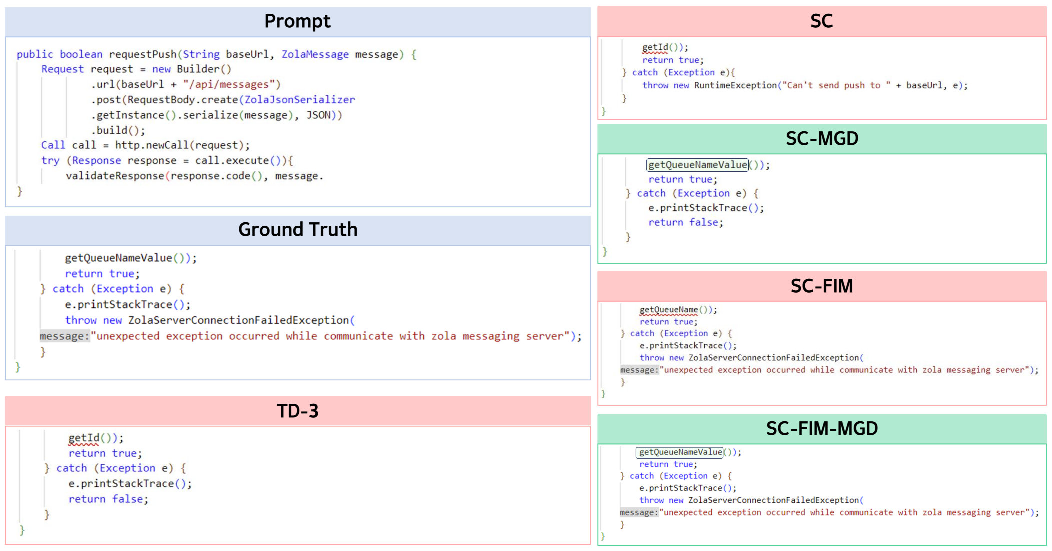
SC

```java
      getProxyUserName().equals(advancedSettings.proxyUserName) ||
      !advancedSettingsComponent.getProxyPassword().equals(advancedSettings.proxyPassword);
}
```

SC-MGD

```java
      getProxyAuthUsername().equals(advancedSettings.proxyUsername) ||
      !advancedSettingsComponent.getProxyAuthPassword().equals(advancedSettings.proxyPassword);
}
```

Figure 9: TD-3 and SC generate code that match the ground truth, except for the identifier names for target objects: `advancedSettings, advancedSettingsComponent`. Augmenting SC with MGD leads to generation of correct identifier names, and therefore agreement with ground truth.



Prompt

```java
public boolean requestPush(String baseUrl, ZolaMessage message) {
    Request request = new Builder()
            .url(baseUrl + "/api/messages")
            .post(RequestBody.create(ZolaJsonSerializer
            .getInstance().serialize(message), JSON))
            .build();
    Call call = http.newCall(request);
    try (Response response = call.execute()){
        validateResponse(response.code(), message.
}
```

Ground Truth

```java
        getQueueNameValue());
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        throw new ZolaServerConnectionFailedException(
    message:"unexpected exception occurred while communicate with zola messaging server");
    }
}
```

TD-3

```java
        getId());
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
```

SC

```java
        getId());
        return true;
    } catch (Exception e){
        throw new RuntimeException("Can't send push to " + baseUrl, e);
    }
}
```

SC-MGD

```java
        getQueueNameValue());
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
```

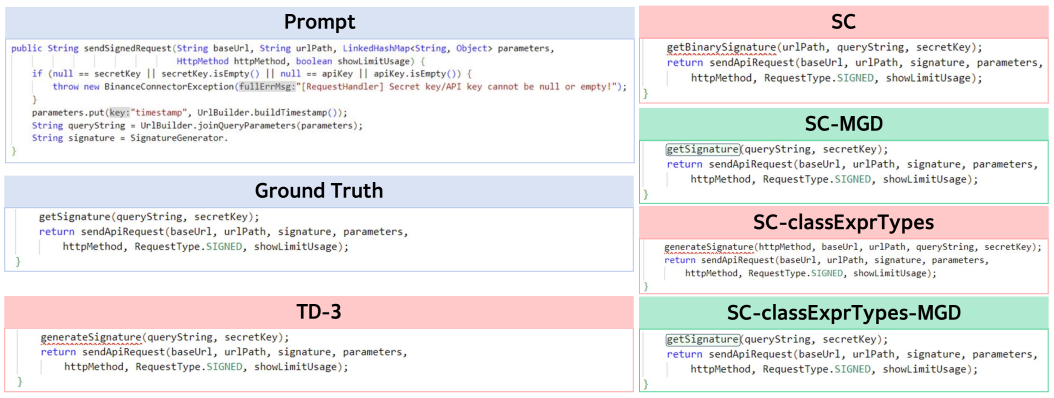
SC-FIM

```java
        getQueueName());
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        throw new ZolaServerConnectionFailedException(
    message:"unexpected exception occurred while communicate with zola messaging server");
    }
}
```

SC-FIM-MGD

```java
        getQueueNameValue());
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        throw new ZolaServerConnectionFailedException(
    message:"unexpected exception occurred while communicate with zola messaging server");
    }
}
```

Figure 10: TD-3 and SC both generate code with invalid identifier names. SC augmented with MGD is able to use the correct identifier name and therefore generates compilable code. However, the generated code does not make use of the defined exception class: `ZolaServerConnectionFailedException` for exception handling. Augmenting SC with Fill-in-the-middle provides the model with the necessary context to perform exception handling, however, the model still hallucinates the identifier name `getQueueName` for the target object `message`. Hence, to get a correct generation, SC is augmented with both FIM and MGD, and this configuration is able to match the ground truth.



Prompt

```java
public String sendSignedRequest(String baseUrl, String urlPath, LinkedHashMap<String, Object> parameters,
                                HttpMethod httpMethod, boolean showLimitUsage) {
    if (null == secretKey || secretKey.isEmpty() || null == apiKey || apiKey.isEmpty()) {
        throw new BinanceConnectorException(fullErrMsg:"[RequestHandler] Secret key/API key cannot be null or empty!");
    }
    parameters.put(key:"timestamp", UrlBuilder.buildTimestamp());
    String queryString = UrlBuilder.joinQueryParameters(parameters);
    String signature = SignatureGenerator.
}
```

Ground Truth

```java
    getSignature(queryString, secretKey);
    return sendApiRequest(baseUrl, urlPath, signature, parameters,
        httpMethod, RequestType.SIGNED, showLimitUsage);
}
```

TD-3

```java
    generateSignature(queryString, secretKey);
    return sendApiRequest(baseUrl, urlPath, signature, parameters,
        httpMethod, RequestType.SIGNED, showLimitUsage);
}
```

SC

```java
    getBinarySignature(urlPath, queryString, secretKey);
    return sendApiRequest(baseUrl, urlPath, signature, parameters,
        httpMethod, RequestType.SIGNED, showLimitUsage);
}
```

SC-MGD

```java
    getSignature(queryString, secretKey);
    return sendApiRequest(baseUrl, urlPath, signature, parameters,
        httpMethod, RequestType.SIGNED, showLimitUsage);
}
```

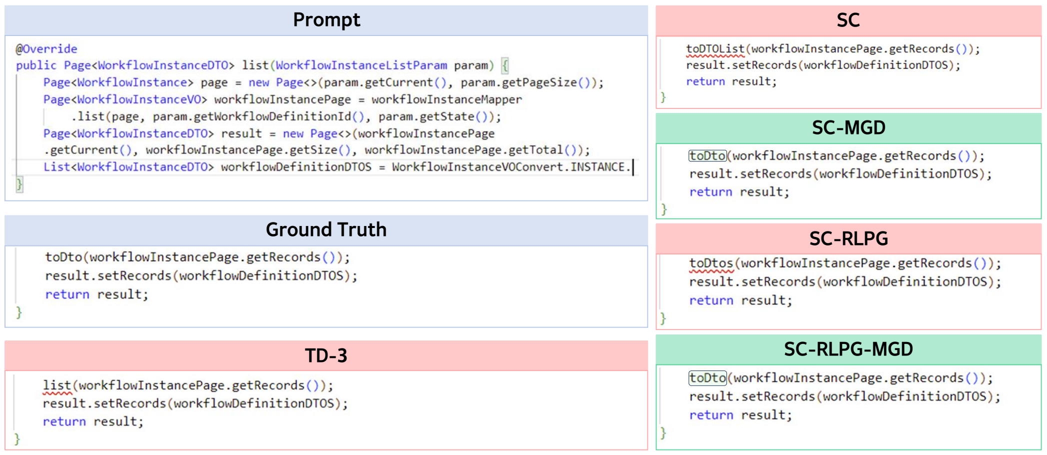
SC-classExprTypes

```java
    generateSignature(httpMethod, baseUrl, urlPath, queryString, secretKey);
    return sendApiRequest(baseUrl, urlPath, signature, parameters,
        httpMethod, RequestType.SIGNED, showLimitUsage);
}
```

SC-classExprTypes-MGD

```java
    getSignature(queryString, secretKey);
    return sendApiRequest(baseUrl, urlPath, signature, parameters,
        httpMethod, RequestType.SIGNED, showLimitUsage);
}
```

Figure 11: Example of generation with classExprTypes prompt augmentation.



Prompt

```
@Override
public Page<WorkflowInstanceDTO> list(WorkflowInstanceListParam param) {
    Page<WorkflowInstance> page = new Page<>(param.getCurrent(), param.getPageSize());
    Page<WorkflowInstanceVO> workflowInstancePage = workflowInstanceMapper
        .list(page, param.getWorkflowDefinitionId(), param.getState());
    Page<WorkflowInstanceDTO> result = new Page<>(workflowInstancePage
    .getCurrent(), workflowInstancePage.getSize(), workflowInstancePage.getTotal());
    List<WorkflowInstanceDTO> workflowDefinitionDTOS = WorkflowInstanceVOConvert.INSTANCE.
}
```

Ground Truth

```
    toDto(workflowInstancePage.getRecords());
    result.setRecords(workflowDefinitionDTOS);
    return result;
}
```

TD-3

```
    list(workflowInstancePage.getRecords());
    result.setRecords(workflowDefinitionDTOS);
    return result;
}
```

SC

```
    toDTOList(workflowInstancePage.getRecords());
    result.setRecords(workflowDefinitionDTOS);
    return result;
}
```

SC-MGD

```
    toDto(workflowInstancePage.getRecords());
    result.setRecords(workflowDefinitionDTOS);
    return result;
}
```

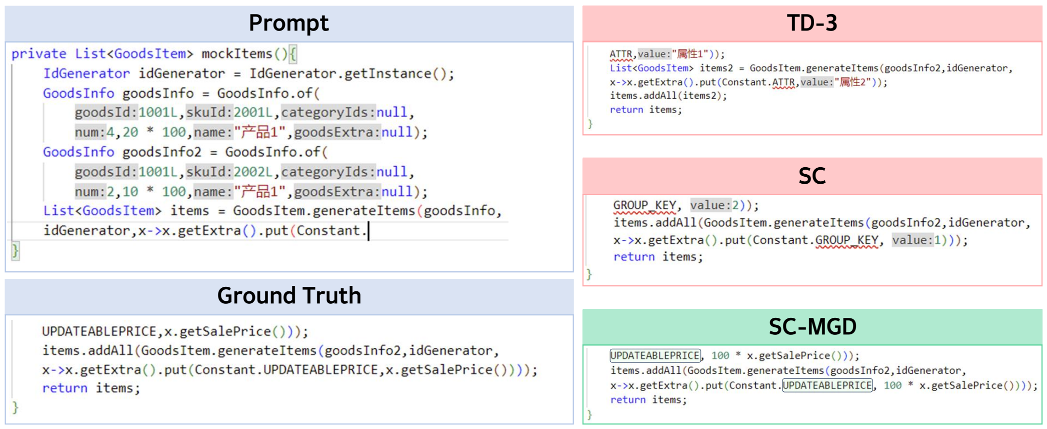
SC-RLPG

```
    toDtos(workflowInstancePage.getRecords());
    result.setRecords(workflowDefinitionDTOS);
    return result;
}
```

SC-RLPG-MGD

```
    toDto(workflowInstancePage.getRecords());
    result.setRecords(workflowDefinitionDTOS);
    return result;
}
```

Figure 12: Example of generation with RLPG prompt augmentation. Augmenting SC with RLPG leads to generation of `toDtos` as compared to `toDTOList`, where both are invalid identifier names for the target object. Augmeting both the configurations with MGD leads to generation of the correct identifier and match with ground truth.



Prompt

```
private List<GoodsItem> mockItems(){
    IdGenerator idGenerator = IdGenerator.getInstance();
    GoodsInfo goodsInfo = GoodsInfo.of(
        goodsId:1001L,skuId:2001L,categoryIds:null,
        num:4,20 * 100,name:"产品1",goodsExtra:null);
    GoodsInfo goodsInfo2 = GoodsInfo.of(
        goodsId:1001L,skuId:2002L,categoryIds:null,
        num:2,10 * 100,name:"产品1",goodsExtra:null);
    List<GoodsItem> items = GoodsItem.generateItems(goodsInfo,
    idGenerator,x->x.getExtra().put(Constant.
}
```

Ground Truth

```
    UPDATEABLEPRICE,x.getSalePrice()));
    items.addAll(GoodsItem.generateItems(goodsInfo2,idGenerator,
    x->x.getExtra().put(Constant.UPDATEABLEPRICE,x.getSalePrice())));
    return items;
}
```

TD-3

```
    ATTR,value:"属性1"));
    List<GoodsItem> items2 = GoodsItem.generateItems(goodsInfo2,idGenerator,
    x->x.getExtra().put(Constant.ATTR,value:"属性2"));
    items.addAll(items2);
    return items;
}
```

SC

```
    GROUP_KEY, value:2));
    items.addAll(GoodsItem.generateItems(goodsInfo2,idGenerator,
    x->x.getExtra().put(Constant.GROUP_KEY, value:1)));
    return items;
}
```

SC-MGD

```
    UPDATEABLEPRICE, 100 * x.getSalePrice()));
    items.addAll(GoodsItem.generateItems(goodsInfo2,idGenerator,
    x->x.getExtra().put(Constant.UPDATEABLEPRICE, 100 * x.getSalePrice())));
    return items;
}
```

Figure 13: TD-3 and SC generate identifier names: `ATTR, GROUP_KEY` respectively for the target class `Constant`, which are both invalid. MGD augmentation leads to generation of correct identifier: `UPDATEABLEPRICE`.