Can Static Analysis Methods Resolve Hallucinations in Open-Domain Code Generation?

Anonymous ACL submission

Abstract

Large Language Model hallucinations in code are more problematic than in regular language, but also potentially more approachable. More problematic because, for example, hallucinated arguments to functions could cause subtle bugs that slow down developers. Potentially more approachable because code can be analysed in ways regular language cannot. We explore the limits of two types of static analysis methods for hallucination detection and mitigation. On natural language-to-code benchmarks that require library use, we find that LLMs generate non-existent library features in 8.1-40% of responses. Our static analysis methods detect many of these issues, but performance varies by LLM and dataset. Manual inspection shows that while there are more cases a static method could plausible catch, a substantial number of cases cannot be caught this way. Overall, this shows that static analysis can reduce hallucinations today, but future work will need fundamentally different approaches to fully solve the problem.

1 Introduction

Hallucination by programming assistant LLMs can lead to code that tries to use non-existent arguments to functions, or entirely made-up functions. Specific forms of this issue have been considered in the past. Spracklen et al. (2025) found that GPT-4 turbo hallucinated packages 4% of the time, while CodeLlama 7B halucinated them 26% of the time. Tian et al. (2025) found that in 10% of cases, GPT-4 mapped data types and structures incorrectly, and in 0.6% of cases, its output did not match external knowledge sources, such as modules' imports. Identifying and fixing these errors creates work for programmers (Tanzil et al., 2024). Using libraries that do not exist can also pose a security risk. An attacker could notice a common hallucination and register a malicious package that matches the hallucination (Spracklen et al., 2025; Krishna et al., 2025).

We investigate the potential for static analysis to detect and mitigate hallucinations in coding output that involves library usage. We propose two methods that inspect the code without executing it and do not require modification of the base LLM. We focus on Python for evaluation, but the core ideas are applicable to any programming language.

Our first approach is grammar-based. We automatically construct a GBNF grammar from the docstrings of libraries and use it in two ways: (1) during generation, to avoid hallucinations with constrained decoding, or (2) after generation, to identify hallucinations and re-prompt an LLM if needed. Previous work has successfully used grammar-based decoding to reduce syntax errors, which is simpler, since the grammar only needs to account for the core language. We extend the approach to avoid misuse of libraries. Our second approach uses off-the-shelf static code analysis tools. This can be used after generation in the same way as the grammar, but not with constrained decoding.

We evaluate on detection and mitigation using three NL-to-code benchmarks that consider library use: DS-1000, Odex and BigCodeBench. While static analysis tools provide more coverage of bugs (up to 82%), grammars do successfully identify some library related errors (up to 15%). Manual analysis shows that there is scope to catch more hallucinations with static analysis, but there are a substantial number of hallucination they are unable to catch. This indicates that resolving the remaining issues will require very different techniques.

It is widely known that static analysis methods can find bugs, but this is the first systematic exploration of how well they address hallucination today and the upper limit of their potential. Our results also demonstrate the impact of benchmark design, as many code benchmarks overlook these issues by not requiring library use or by lacking a clear description of natural language requests that does not match their test cases.

2 Related work

Hallucinations in code Tian et al. (2025) define code hallucination as code generated by LLMs that might be syntactically and semantically plausible but cannot execute or meet requirements. They consider code errors as a type of code hallucination. Our definition, described in the next section, is slightly different. We do not consider all errors to be hallucinations. Existing work in AI on addressing code hallucinations uses approaches such as retrieval-augmented generation (RAG), iterative grounding, few shot prompting, and fine-tuning (Eghbali and Pradel, 2024; Liu et al., 2024a; Agarwal et al., 2024; Mok et al., 2024; Li et al., 2023). Although these approaches sample better quality tokens, they differ from our work by not addressing detection and mitigation simultaneously (Tanzil et al., 2024), and cannot guarantee that hallucinations have been resolved. There has been related work in Software Engineering on Automatic Program Repair (APR), but this is a more general challenge (Xia et al., 2023; Koutcheme et al., 2023; Prenner et al., 2022; Liu et al., 2024b; Zhang et al., 2023; Wuisang et al., 2023; Ni et al., 2024). Our research aims to mitigate hallucinations by detecting and constraining them before the code is executed and has an error.

Hallucinations and external knowledge sources of code LLMs pre-trained on specific tools produce inconsistencies and hallucinations in APIs (Roy et al., 2024), and LLMs optimized for code produced higher rates of package hallucinations (Krishna et al., 2025). Ayala and Bechard (2024) explored finetuning on JSON-structured workflows with annotated suggestions. When they compared a model with and without augmentation, they observed a loss in diversity and an increase in hallucinations. Eghbali and Pradel (2024) proposed a RAG approach for factual open-domain code, which iteratively queries an LLM with API references as context. RAG reduced hallucinations by 3%, and the iterative approach led to a 15% improvement in matching the exact imports, though not necessarily accounting for the code being executable.

Static Analysis in code generation Ding et al. (2023) suggests that static program analysis from generated code can reduce errors and resources when evaluating code generation with execution-based benchmarks. Jaoua et al. (2025) showed that RAG with knowledge bases and static code analysis can be a cost-efficient method for code reviews. However, on bug detection, Chen et al. (2025a) compared static analysis tools with finetuned and pre-trained bug detection models and found that when the tool and the model were used together, recall increased, and on annotated programs, precision increased as well. This shows the importance of well-annotated packages for accurate static analysis. We relied on static code analysis to prevent faulty code generation, while instructing the LLM to repair bugs detected by these tools.

Inference based solutions In package hallucination, Spracklen et al. (2025) demonstrated that hallucinations are not due to sampling, as they still occur with greedy inference. Fu et al. (2024) used constrained decoding to produce more secure code, which is different from our goal, but supports the feasibility of our approach. Roy et al. (2024) perform constrained decoding to check a conversation intent and factuality in self-created APIs, but not in real-world use. Chen et al. (2025b) tested API use, but did not evaluate execution, nor requests from natural language. While they use library aliases as prefixes, when compared to our approach, using a grammar can provide additional validation on syntax and semantics.

Grammar-constrained decoding Focusing on errors, Olausson et al. (2023) concluded that the effective rate of code repair directly relates to the quality and size of the LLM. In contrast, Geng et al. (2023) found that grammar-constrained decoding, without finetuning and with scarce data, boosts any size of LLM on structured tasks defined through formal grammar. Ugare et al. (2024) used context-free grammars to address syntax errors, reducing them by 96% on Python and Go, which clearly demonstrates the potential of this approach to resolve issues in output. Similar to our work, SYNCHROMESH (Poesia et al., 2022) explored grammar constraints on SQL and JSON output, derived from samples. While flexible, that has the disadvantage that it is not possible to guarantee consistency with an external library.

Benchmarks Benchmarks on code hallucinations (Hallucode; Liu et al. 2024a; CodeMirage; Agarwal et al. 2024) treat this problem as a classification task where the language model needs to detect the type of hallucination from a given snippet of code. They tend to focus on logical errors or inconsistency with the request, whereas we are interested in library use, leading us to evaluate with open-domain code execution benchmarks.

3 Method

We define code hallucination as tokens produced by an LLM that do not match the context of the request. By 'context', we mean the available libraries, other code files in the codebase, and the prompt provided by the user. For example, generating a function name that does not exist in a library is a hallucination. Note that not all errors are hallucinations, e.g., syntax errors are not.

To detect hallucinations, we consider one approach using a grammar and one using static code analysis tools. We also compare with a simple baseline: asking o3-mini to judge whether code will be executable.

3.1 Approach 1: Automatically Extracted Grammars

Our first approach uses automatically extracted grammars to detect hallucinations. We developed a method to analyse the documentation for a library and automatically construct a grammar that will only accept code that matches the library definition. Our grammar is in GBNF syntax[1] since that can be directly used with *llama.cpp* to constrain decoding.

3.1.1 Grammar creation

Our grammar has three core symbols:

`Except-library`: accepts any token aside from the library's alias and name, and the keywords "import" and "from".

`Library`: uses the library name or alias as a prefix for built-ins, initializers, and constants.

`Import`: uses the keywords "import " and "from " as a prefix to validate packages in the environment of the benchmark and Python built-in modules.

These allow us to handle any output not related to libraries, including code and even natural language. This is useful as it allows us to focus on identifying issues with library use and it allows for the style of output where there is a brief explanation, followed by a code snippet, and then further explanation in natural language.

We construct the rest of our grammar in two steps. First, we use the *inspect* module to retrieve the target library's documentation and represent the key information in JSON. Second, we combine information from the JSON data with Python's core language specification[2] to create a single EBNF grammar that covers library use in the context of other code. Each rule accounts for one of the library functions, with variations to account for variations such as optional and repeated arguments. A two-step approach provides modularity that could make adaptation to other languages easier in future.

Almost all of this processing is automatic. The Python language formal grammar was automatically converted in 4 hours of work. The library grammars were automatically extracted. Common aliases for libraries (e.g., np for numpy) were automatically extracted from use in the benchmark with a regular expression. To expand to more libraries, defining common aliases is the only step that requires validation. One limitation of our approach is that it cannot handle atypical aliases because llama.cpp does not allow the grammar to track state. However, atypical aliases are rare and likely to become even more rare as LLM use rises and so the same alias is consistently proposed.

Listing 1 shows a slightly simplified snippet of Numpy's grammar. We highlight in bold the main symbols that verify a program with the grammar. Starting from the symbol **root**, we can accept all tokens that describe instructions in natural language, except for the prefix of the library name *numpy* or alias *np*. If a library prefix is matched, all the following tokens should also match those from the **library** symbol, e.g., numpy methods, builtins, or constants. Similarly, matching the token *import* will add as next states the **import-from** and **import-name** symbols, forcing the verification of subsequent tokens as Python's builtins and environment modules.

3.1.2 Grammar-constrained decoding

We use the grammar to constrain the output of the LLM to be consistent with what the grammar permits. During inference, we constrain the probability distribution of the next token to only have positive values for tokens that are valid according to our grammar. Which tokens can be positive is determined by a parser that matches partial sequences with the grammar we provide. This approach has the advantage that it only filters out invalid outputs, with no impact on scoring for valid outputs.

We use the parser and decoding integration provided by *llama.cpp*. This has the advantage that our approach is compatible with any of the open-source LLMs hosted in Ollama. To be efficient, *llama.cpp* actually samples without filtering the space of output tokens, and if the sample is ac-

[1] This is a variant of EBNF developed for llama.cpp that adds some features from regular expressions.

[2] This is available in EBNF and so we converted it to GBNF.

```
# Connector rules between the library and the GBNF
root ::= ( except-library | library | import )+
except-library ::= ( "numpy" [^.] | "np" [^.] | "from" [^ ] | "import" [^ ] )
library ::= ( "numpy." | "np." ) ( np-numpy-methods | np-numpy-builts | np-numpy-const )
class-name ::= ( "n"[^p] | "nump"[^y] )
class ::= class-name (trailer)*
# Import statements
import ::= ( import-name | import-from ) "\n"
import-from ::= "from " dotted-name " import " name ( " as " name)?
import-name ::= "import " dotted-name (" as" name)?
dotted-name ::= ( python-modules | environment-modules )
# Built-ins and constant rules
np-numpy-builts ::= "char" | "compat" | "compat.py3k" | "compat.tests" | "core" ...
np-numpy-const ::= "e" | "euler_gamma" | "inf" | "nan" ...
np-numpy-methods ::= np-numpy-copyto | ...
```

Figure 1: Python's partial GBNF with a subset of rules defining the *numpy* library.

cepted by the grammar it continues without needing to compute the mask over the output token options. To further improve speed, we implemented caching the states from the non-deterministic pushdown automata used by *llama.cpp*. We conducted our experiments on a local computer with an M2 processor, 8-core CPU, 10-core GPU, and 24GB RAM.

3.2 Approach 2: Off-the-Shelf Static Code Analysis

Our second approach is to use off-the-shelf Static Code Analysis tools. Specifically, we consider *mypy* (an explicit type annotation tool) and *pyright* (a type inference tool). These were developed to detect errors in human-written Python code. They are more general than our grammar approach, but are also more language-specific. We configured both tools with automatically generated type stubs (van Rossum et al., 2014) to provide information about the functions in the standard library and third-party libraries. Providing these annotations is critical for identifying the library features for our use case. For each error these tools identify, we use the tools' analysis output as guidance for the LLM to repair the code.

4 Experiments

In this section, we describe our experimental set up, then consider detection, followed by mitigation.

Benchmarks Recent benchmarks developed to target hallucinations in code (Collu-Bench; Jiang et al. 2024; CodeHalu; Tian et al. 2025, APIHulBench; Chen et al. 2025b) either lack NL descriptions or evaluate with an exact match, rather than using test cases. In this work, we are concerned with hallucinations in code as a response to instruction prompts related to library usage. We evaluate with three open-domain code execution benchmarks: DS-1000 which has 1,000 problems over 7 libraries (Lai et al., 2023), Open-Domain EXecution-based natural language (ODEX), which has 945 problems in four languages (en, es, ja, ru) over 79 libraries (Wang et al., 2023) and BigCodeBench (instruct) which has 1,140 problems over 139 libraries (Zhuo et al., 2025)[3]. In all of them, answers involve multiple libraries, either explicitly specified in the request, or implicitly needed. Rather than using grammars that cover all libraries for every question, we use the ones needed: explicitly indicated ones and common related libraries.

Metrics For detection, we consider all errors and measure precision and recall. We also introduce *Imaginary Features Recall (IF R)*, which focuses on hallucinations by measuring the percentage of errors that are one of four types: attribute, import, type [4], and module not found. For mitigation, we use three metrics. *Pass@k (P@k)* evaluates functional correctness and returns the ratio of k samples passing a set of test cases (Haque et al., 2023)[5]. *Execution Rate (ER)* measures how frequently code executes, without considering correctness. *IF* is a measure of the reduction in imaginary features. For examples of these errors, and which token causes the issue, see Table 6 in the Appendices.

[3]We used BigCodeBench under the Apache 2.0 license, and DS-100 and ODEX under cc-by-sa-4.0.

[4]We think type errors are important to measure since an LLM might generate nonexistent parameters in functions, or map incorrect data types to function calls.

[5]We evaluate using Pass@1; so the LLM has a single run to give the correct answer.

	claude-3			gpt-3.5			gpt-4			IBM-Granite		
	OP	OR	IF R	OP	OR	IF R	OP	OR	IF R	OP	OR	IF R
DS-1000												
Mypy	0.22	0.27	0.22	0.26	0.34	0.23	0.16	0.23	0.26	0.63	0.26	0.21
Pyright	0.61	**0.70**	**0.85**	0.57	**0.60**	**0.75**	0.50	**0.62**	**0.84**	**0.95**	**0.55**	0.52
o3-mini	**0.63**	0.43	0.29	**0.61**	0.45	0.29	**0.55**	0.43	0.26	0.79	0.51	**0.68**
Grammar	0.36	0.14	0.16	0.45	0.11	0.15	0.53	0.15	0.20	0.57	0.10	0.17
Odex												
Mypy	**0.85**	0.15	0.17	0.39	0.66	0.60	0.43	0.81	0.70	**0.81**	0.18	0.17
Pyright	0.76	**0.22**	**0.26**	0.39	**0.68**	**0.64**	0.43	**0.82**	**0.71**	0.72	**0.34**	**0.36**
o3-mini	0.61	0.09	0.08	**0.61**	0.22	0.11	**0.79**	0.32	0.08	0.74	**0.34**	0.34
Grammar	0.35	0.08	0.11	0.43	0.08	0.10	0.52	0.11	0.12	0.59	0.15	0.17
BigCode												
Mypy	0.16	0.05	0.03	0.22	0.07	0.03	0.24	0.09	0.08	0.47	0.09	0.10
Pyright	0.32	**0.20**	**0.24**	0.28	**0.16**	**0.14**	0.34	**0.20**	**0.24**	0.62	0.30	0.36
o3-mini	**0.64**	0.12	0.15	**0.56**	**0.16**	**0.14**	**0.57**	0.11	0.14	**0.77**	**0.48**	**0.50**
Grammar	0.19	0.04	0.06	0.24	0.12	0.11	0.23	0.06	0.06	0.45	0.09	0.06

Table 1: Detection results for all query models, detection methods, and datasets. We defined *OP* as the overall precision and *OR* as the overall recall when detecting all types of bugs. While *IF R* represents recall only on imaginary features. We do not show IF precision as it was 1 in all cases, except for o3-mini in IBM-Granite on DS-1000, which was 0.98.

Generation Models We test our approaches on the output of four LLMs: Claude-3, GPT-4, and GPT-3.5, and IBM-Granite 3B. For the API based models, we used a temperature of one, and for the open source model, we used 0.4. These values are as reported in prior work using these models on these benchmarks. We chose IBM-Granite because of its high scores on DS-1000. The model is fine-tuned for tasks such as code fixing and explanation (Mishra et al., 2024). We validated samples for all 4 models on all questions in DS-1000 and BigCodeBench, and 550 questions from Odex (not the full dataset because we skipped questions that do not require library imports).

4.1 Detection

Table 1 presents precision and recall for our methods and the baseline. We see three distinct patterns of results. The grammar has the lowest recall, reflecting the focus on library related errors, which are a small subset of the errors. The baseline, o3-mini, has high precision and low recall. Mypy and Pyright, the off-the-shelf static analysis tools, have very similar results, with higher recall than precision, while Pyright better balances between the two than o3-mini most of the time. However, this depends on the benchmark.

One key issue with the grammar approach is the limited information in docstrings. For example, docstrings do not define namespaces, and so `int32` in `result = tf.random.uniform(..., dtype=tf.int32)` is marked as an error. Another limitation is that the grammar must match a library name or alias on every call. To see the issue this causes, consider `cv = CountVectorizer(stop_words='english', punctuation_pattern=r'\\W')`. This code uses `CountVectorizer`, which has been imported using `from ... import CountVectorizer`. `punctuation_pattern` is a hallucination, but the grammar does not identify it because this code doesn't use the library name or alias in the call. We could modify the grammar to not require explicit use of the library name or alias, but then we would have additional false positives.

	Caught	TP	Overlap	Capable
DS-1000				
Static	56.7	37.6	17.9	77.0
Grammar	11.2	7.9	85.7	18.0
Odex				
Static	19.0	4.0	12.5	70.5
Grammar	6.5	2.5	20.0	10.5
BigCode				
Static	23.0	10.0	5.0	48.5
Grammar	8.5	5.0	100.0	7.0

Table 2: Percentage of overall errors that these tools are able to catch and are capable of, if modified.

To further understand these results, we manually inspected a sample of 200 failure cases for Imaginary Features on each benchmark, with an even sample size across models. We labeled them according to whether we believed a static analysis

approach could feasibly catch them, and why or why not this is possible due to their root cause. A description of these categories is in Section A.3.

Table 2 *Caught* column shows whether the tools were able to detect the error, and the true positive column *TP* if this error was detected by the right reason. In all of them, we observe a high rate of False Positives (FP) when manually examining why the error was generated. Due to the FP, we observe variability in the overlap between the detected samples. However, in Odex, this is when the grammar is far from static. This is particularly evident in the datetime library, where the way the library is imported collides with the built-in and module definitions, resulting in high precision and recall for the grammar, but opposite to static tools.

One important clarification to make is that the *Capable* column measures whether the tool, due to its characteristic, is able to detect an error, not necessarily to solve it. We believe that the static tools have room for improvement, primarily in the Data and Control Flow operations. For the first case, depending on the problem and library information, datatypes might mutate during the program. For example, when given a dataframe as input and calculating the average of a single column, it returns a series. However, in the case of two columns, a dataframe is returned. Therefore, static tools should consider these cases as part of their data flow. In the case of control flow, the benchmarks work on their test pipeline by defining a function that will be automatically run in the test suite later. In several cases, the static tool cannot access the code inside the function and therefore misses analyzing this code. Another instance of this type of error occurs in Lambda functions, within an apply or map function; the tools skip this nested flow, and if implemented, it will increase detection in the DS-1000 benchmark, since this is a standard for data science libraries using dataframes.

In the Table 2, we see that static tools, in Odex, are capable of detecting errors in 70.5%; however, this is because many cases of Odex do not return anything in the function, resulting in returning 'None' being an easy case for the Static tools to identify. However, this is not the case for BigCode, with a more diverse source of errors.

For each error in the sample, we also annotated the cause and whether it happened during the LLM's code or in code while processing the test. These results yield insights into what errors are feasible to solve.

There were three types of issues we considered infeasible, all of which related to prompt specification matching the test cases. (1) When a calculation requires the input data to design the code. For example, consider a column with mixed data types. (2) When the request does not specify the input or output data type, or the prompt is ambiguous about the library it uses. (3) Execution errors that occur in testing code and are not part of the generated code.

Table 3 presents categories of the source of errors in the sample of each benchmark in percentage. We consider errors that are feasible to detect as those related to Control Flow and Data Flow (*Flow*) in static tools, as well as rule definitions that match the *Library* definitions. These two represent the majority of cases in DS-1000. However, for Odex and BigCode, we have a more diverse distribution. For our infeasible case 1, errors that depend directly on the knowledge of the LLM, such as setting a variable to a negative value without reason and then encountering an exception because it requires positive numbers, are definitely outside the scope of the tools.

The remaining cases are independent of the LLM or the static analyzer and depend entirely on a well-designed benchmark in both prompt and test cases. For case 2, an execution error may occur during the generated code if the prompt does not specify the type of input. A common case in BigCode is using the keyword 'Dataset'. While the LLM designs code for a dataframe, it evaluates the function using NumPy array as input and expects a NumPy array as output. Another example is test cases containing 'None' in the data. While some prompts specify the requirement to consider data that might contain 'None' values, others do not, and have a test case that evaluates this characteristic. These two error causes in DS-1000 represented 1.1%, while in Odex 36.9%, and in BigCode (instruct) 34.1%. We correlate these to the prompt character count; DS-1000 has almost 800 more characters than Odex and almost 200 more characters than BigCodeBench.

For case 3, we observed that the three benchmarks contain test cases that fail during the initialization of input data. We labeled these as *Test case error* and found that they appear 2.8-6.5% of the time. Finally, with benchmarks that test the breadth and depth of library function calls, we consider *Test breadth* as whether the assertion cases have different ways to query the library from a valid solution. An example is when asking to set

Error cause	DS-1000	Odex	BigCode
Generation			
Flow	57.1	42.4	34.2
Library	14.7	6.6	6.0
Data-Logic	8.5	6.1	7.5
Ambiguous input	0.0	6.6	7.5
Test			
Test case error	2.8	4.0	6.5
Ambiguous output	1.1	30.3	26.6
Test breadth	15.8	4.0	11.6
Prompt (Char count)	871.8	87.5	663.2

Table 3: Cause of errors in the sample on each benchmark in percentage.

```
# make a stem plot of y over x
# and set the orientation to be horizontal
plt.stem(x, y, use_line_collection=True)
plt.show()
```

```
# make a stem plot of y over x
# and set the orientation to be horizontal
plt.stem(x, y, linefmt='C0-', markerfmt='o',
basefmt='C0-')
plt.xlabel('x')
plt.ylabel('y=e^{sin(x)}')
```

Figure 2: Example of how constrained decoding impacts output. **On the top.** Unconstrained response with imaginary feature. **On the bottom.** Constrained response with factual parameters.

a title in a plot; there are several ways to evaluate this. One could use either `ax.get_title()` or `plt.gca().get_title()` depending on how the LLM sets the title; however, these three benchmarks only consider one of those options. In reality, we need to consider different valid options in the library to account for diverse valid solutions.

4.2 Repair

In this section, we turn from detecting errors to repairing them. We evaluate on the examples that contained an error when executed: 1383 samples in DS-1000, 1171 in BigCode, and 825 in Odex (across all four generation models). To repair the error, we prompt an LLM with the code, the error, and a request to resolve the issue. We try o3-mini combined with static analysis tools, or no tool at all. We average over three runs.

Table 4 shows that our approach consistently improved all metrics, particularly execution rate (ER) and Imaginary Features (note that for IF, lower is better). Interestingly, o3-mini performs better without information from static analysis on DS-1000 and BigCodeBench, but not on Odex; these results might be related to the quality of the benchmarks.

4.3 Mitigation

Finally, we consider avoiding generation of errors entirely by using constrained decoding. This can only be done with the grammar approach and an open-source model. Listing 2 shows an example of how the grammar can help. Without constrained decoding, the LLM produced an imaginary parameter **'use_line_collection'**, but with constrained decoding it generates valid parameters. Table 5 shows that the benchmarks resulted in a lower rate of imaginary library features when the model was constrained in DS-1000, but showed moderate improvement in Odex, and no improvement in BigCodeBench [6]. The results for ER and P@1 are more variable. All shifts are small, which is consistent with the low detection rate of the grammar-based approach.

Why did *Pass@1* decrease? This negative result relates to the difficulty of getting accurate information from docstrings. When the correct answer is not defined in the grammar, the constrained output may be wrong. This is consistent with previous work on extracting hyperparameter constraints from machine learning operators, which found low precision in docstrings and proposed other methods like weakest-precondition (Rak-Amnouykit et al., 2021). Another case is when the LLM has memorized an answer or only knows one approach to the problem; even plausible candidates in the grammar might not appear in LLMs' pretraining data. Recent work on hallucination benchmarks (Ravichander et al., 2025) suggests that coding tasks, such as library hallucinations, appear on pretraining data examples; while being true in a specific document, in isolation, they lose context and lead to factual errors.

In a time comparison, we observed that an unconstrained response takes an average of 22.3 seconds, while a constrained one takes an average of 34.76 seconds. We also evaluated *Sampling cost* (Olausson et al., 2023). This is the total number of tokens sampled from the model. We found that the evaluation on Odex resampled half of the times compared to DS-1000. In DS-1000, the constrained approach

[6] In Table 4 we used IBM-Granite's responses from the BigCodeBench Leaderboard; however, there was no specification of the IBM-Granite version, so we re-run IBM-Granite with our version to do a fair comparison with our constrained model.

	claude-3			gpt-3.5			gpt-4			IBM-Granite		
	ER	P@1	IF	ER	P@1	IF	ER	P@1	IF	ER	P@1	IF
DS-1000												
o3+Self	**82.3**	**44.3**	**2.7**	**82.8**	**37.9**	**2.6**	**85.3**	**49.3**	1.9	**56.3**	**12.0**	**5.2**
o3+Mypy	80.0	43.7	2.9	80.3	37.7	3.3	82.5	49.1	**1.7**	45.5	10.1	8.1
o3+Pyright	80.3	**44.3**	3.2	79.9	37.4	3.5	82.1	48.7	2.4	44.5	10.1	8.6
No repair	73.1	42.7	9.1	73.2	36.6	9.7	76.8	47.7	8.1	33.6	8.9	11.4
Odex												
o3+Self	70.3	43.6	16.6	67.3	40.4	15.8	67.5	**43.2**	16.0	**50.3**	**19.6**	**23.0**
o3+Mypy	71.1	**44.2**	**16.2**	67.3	40.2	16.8	67.5	43.0	15.6	48.5	19.0	24.2
o3+Pyright	**71.3**	44.0	**16.2**	**67.7**	**40.6**	16.4	**68.1**	**43.2**	16.0	**50.3**	19.4	23.4
No repair	66.9	42.2	20.4	59.6	35.8	15.8	57.0	35.2	13.7	46.3	18.0	24.0
BigCode												
o3+Self	**82.6**	**47.3**	29.8	**82.1**	**41.3**	**34.3**	**82.6**	**47.2**	35.4	**77.7**	**26.7**	39.0
o3+Mypy	79.0	45.5	30.5	77.9	39.4	35.7	80.0	46.1	36.4	62.2	21.1	39.0
o3+Pyright	80.0	45.9	**28.9**	78.3	39.4	36.4	80.4	46.1	**34.5**	66.7	23.2	**38.2**
No repair	79.0	45.5	30.1	77.6	39.3	35.7	79.6	46.0	36.6	61.0	20.6	40.0

Table 4: Code performance with various repair methods.

	ER	P@1	IF
DS-1000			
Unconstrained	33.6	**8.9**	11.4
Constrained	**36.3**	6.5	**8.0**
Odex			
Unconstrained	**46.3**	18.0	24.0
Constrained	45.9	**18.6**	**23.0**
BigCode			
Unconstrained	**58.1**	**16.0**	**44.1**
Constrained	54.2	12.7	46.6

Table 5: Overall benchmark results with and without constrained generation using our grammar.

added around three to eight samplings per request, while in Odex, one to three samplings. This could be explained by the distribution of rules in each grammar, since in DS-1000, we found more uniform and larger grammar rules, as seen in Figure 4 in the Appendix.

5 Discussion

Melcer et al. (2024) combined lexer functionality in the left-most parser to account for code indentation; these *INDENT and DEDENT* grammar constraints were added in the right quotient. They suggest their approach could be used with static analysis for verifying partial programs; this idea was also explored by Ding et al. (2023) in Python, and by Mündler et al. (2025) in Typescript code; however, as seen from our detection experiment, more work needs to be done on the inference of types in dynamic type languages without annotations such as Python.

Previous work on grammar-constrained decoding suggests API calls as an avenue for these methods (Koo et al., 2024; Poesia et al., 2022), while our proposed grammar can identify when the library is being used through an alias or name, our method is less precise since it cannot keep track of scopes and code's data flows; using code flow graphs like in Static Analysis is something to look after. Integrating Static Analysis functionality into the decoder to behave as a parser and lexer could be a future direction for constrained decoding in code. However, these tools are far from perfect when analyzing machine code. Additionally, from our analysis in Section 4.1, we require more standards for natural language annotations to match test cases, as well as broader test suites for executing code benchmarks to account for different valid solutions, so that we can evaluate with fairness to LLM's capabilities.

6 Conclusions

We explore the potential for static analysis methods to address hallucination in code. This work is the first to use grammar-constrained decoding to prevent LLMs from generating code that uses imaginary library features. Results were poor for repair and mitigation, but even as detectors, these methods can be useful, since they can alert users about issues. Both methods were affected by unambiguous requests that influenced the output quality and, therefore, the detection tool's performance. There is still scope for futher improvement, but our manual analysis indicates there are a significant number of hallucinations they will not be able to resolve. Further work on detecting issues will need to apply significantly different methods.

7 Limitations

One of the main limitations in our approach is mining docstrings, since the quality of the grammar depends on how well these are retrieved. As mentioned in Section 4.1, docstrings do not define namespaces and aliases, and ctype libraries have less well-defined docstrings. Another limitation of this work is that we only focused on and analyzed the first error in execution; we did not track whether our method solved this same error or encountered a new one of the same type in another line of code. Although we benchmark several LLMs in different scenarios, we did not run more than one test in some experiments. Another limitation in the analysis of the Section 4.1, is that we used a sample; the percentages we show are a summary of the sample. We are unsure whether the ratios might not hold for the full dataset; however, we are making the annotations publicly available so that more researchers can either agree with or refute our work.

On another subject, our approach explores the memoization of parsing states as an optimization for grammar-constrained decoding. Other approaches, such as Domino (Beurer-Kellner et al., 2024), have optimized grammar-constrained decoding by unifying the transverse of precomputing subterminal trees. In Syncode (Ugare et al., 2024), they precompute this set through the union of boolean masks, highlighting the importance of grammar-constrained decoding optimization for its adoption in structure but non-deterministic tasks.

As mentioned by Geng et al. (2023), abstract vs. concrete syntax problem occurs on LLMs trained with different tokenization methods like Byte-Pair Encoding (BPE) since the token can be tokenized in more than one way and reject plausible candidates; for example, in our use case, the LLM might tokenize *pd.DataFrame* as *pd. ... Data ... Frame*. Adapting methods such as the one Koo et al. (2024) proposed by which detokenize characters into tokens and back to text using Finite-state transducers (FST) could decrease sampling cost by accepting partial tokens and decrease time through transversing multiple symbols in the PDA at once; this means reducing the number of next states to parse. Given the similarity of their method to current *llama.cpp* implementation, this is a viable direction for our work.

References

Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2024. Codemirage: Hallucinations in code generated by large language models. In *2nd IJCAI Workshop on no-code copilots Automates*, Korea.

Orlando Ayala and Patrice Bechard. 2024. Reducing hallucination in structured outputs via retrieval-augmented generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 228–238, Mexico City, Mexico. Association for Computational Linguistics.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding LLMs the right way: Fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 3658–3673. PMLR.

Boqi Chen, José Antonio Hernández López, Gunter Mussbacher, and Dániel Varró. 2025a. The power of types: Exploring the impact of type checking on neural bug detection in dynamically typed languages. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25.

Yujia Chen, Mingyu Chen, Cuiyun Gao, Zhihan Jiang, Zhongqi Li, and Yuchi Ma. 2025b. Towards mitigating api hallucination in code generated by llms with hierarchical dependency aware. *Preprint*, arXiv:2505.05057.

Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. A static evaluation of code completion by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 347–360, Toronto, Canada. Association for Computational Linguistics.

Aryaz Eghbali and Michael Pradel. 2024. De-hallucinator: Mitigating llm hallucinations in code generation tasks via iterative grounding. *Preprint*, arXiv:2401.01701.

Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. 2024. Constrained decoding for secure code generation. *Preprint*, arXiv:2405.00218.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, Singapore. Association for Computational Linguistics.

Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. Fixeval: Execution-based evaluation of program fixes

for programming problems. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18. IEEE.

Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. 2025. Combining large language models with static analyzers for code review generation. In *22nd IEEE/ACM International Conference on Mining Software Repositories, MSR 2025, Ottawa, Canada, April 28-29, 2024*. ACM.

Nan Jiang, Qi Li, Lin Tan, and Tianyi Zhang. 2024. Collu-bench: A benchmark for predicting language model hallucinations in code.

Terry Koo, Frederick Liu, and Luheng He. 2024. Automata-based constraints for language model decoding. In *First Conference on Language Modeling*.

Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. 2023. Automated program repair using generative models for code infilling. In *Artificial Intelligence in Education*, pages 798–803, Cham. Springer Nature Switzerland.

Arjun Krishna, Erick Galinkin, Leon Derczynski, and Jeffrey Martin. 2025. Importing phantoms: Measuring llm package hallucination vulnerabilities. *Preprint*, arXiv:2501.19012.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024a. Exploring and evaluating hallucinations in llm-powered code generation. *Preprint*, arXiv:2404.00971.

Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2024b. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.*

Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. 2024. Constrained decoding for fill-in-the-middle code language models via efficient left and right quotienting of context-sensitive grammars. *Preprint*, arXiv:2402.17988.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, and 27 others. 2024. Granite code models: A family of open foundation models for code intelligence. *ArXiv*, abs/2405.04324.

Jisoo Mok, Mohammad Kachuee, Shuyang Dai, Shayan Ray, Tara Taghavi, and Sungroh Yoon. 2024. LLM-based frameworks for API argument filling in task-oriented conversational systems. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 419–426, Mexico City, Mexico. Association for Computational Linguistics.

Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-aware constraining for code LLMs. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: teaching large language models to reason about code execution. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*.

Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can openai's codex fix bugs? an evaluation on quixbugs. In *Proceedings of the Third International Workshop on Automated Program Repair*, APR '22, page 69–75, New York, NY, USA. Association for Computing Machinery.

Ingkarat Rak-Amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. 2021. Extracting Hyperparameter Constraints from Code. In *ICLR Workshop on Security and Safety in Machine Learning Systems*, Virtual, United States.

Abhilasha Ravichander, Shrusti Ghela, David Wadden, and Yejin Choi. 2025. HALoGEN: Fantastic LLM hallucinations and where to find them. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1402–1425, Vienna, Austria. Association for Computational Linguistics.

Shamik Roy, Sailik Sengupta, Daniele Bonadiman, Saab Mansour, and Arshit Gupta. 2024. FLAP: Flow-adhering planning with constrained decoding in LLMs. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 517–539, Mexico City, Mexico. Association for Computational Linguistics.

Joseph Spracklen, Raveen Wijewickrama, A H M Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadliwala. 2025. We have a package for you! a comprehensive analysis of package hallucinations by code generating llms. In *SEC'25: Proceedings of the 34th USENIX Conference on Security Symposium*. USENIX Association.

Minaoar Hossain Tanzil, Junaed Younus Khan, and Gias Uddin. 2024. Chatgpt incorrectness detection in software reviews. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(1).

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Syncode: Llm generation with grammar augmentation. *Submitted to Transactions on Machine Learning Research*. Under review.

Guido van Rossum, Barry Warsaw, and Nick Coghlan. 2014. Style guide for Python code - type hints. PEP 484.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290, Singapore. Association for Computational Linguistics.

Marchel Christhoper Wuisang, Marcel Kurniawan, Komang Andika Wira Santosa, Alexander Agung Santoso Gunawan, and Karen Etania Saputra. 2023. An evaluation of the effectiveness of openai's chatgpt for automated python program bug fixing using quixbugs. In *2023 International Seminar on Application for Technology of Information and Communication (iSemantic)*, pages 295–300.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions.

A Appendix

A.1 Risks

We do not see any significant risk introduced by this work.

A.2 Imaginary features

In Table 6, we present an analysis of the generated code and highlight the tokens that we consider to be imagined by an LLM in the library usage context, and that will result in an error when the code is executed.

A.3 Annotation categories

Table 7 contains the categories chosen for annotating the sample of errors of Imaginary Features. The full annotation guideline can be found under the supplemental material of this work.

A.4 Repair

In Figure 3, we compare the time taken for each method to detect a bug. *Mypy* was the fastest detection tool with a mean of 1s, followed by *Pyright* and grammar with an average of 2.5s, and o3-mini with a mean of 4s

A.5 Mitigation

Figure 4 shows the distribution of the number of rules used in each question between the benchmarks, and Figure 5 compares the distribution of resample tokens for each benchmark. Figure 6 shows the distribution of the time taken between the constrained and unconstrained versions of IBM-Granite in BigCodeBench.

A.5.1 Memoization

As seen in Figure 7, we found no gains in time spent per token decoded. Further analysis is needed on the latency generated by Ollama and resampling. As for the remaining work on memory usage on pushdown states in both methods.

Imaginary Features	Description
TypeError	A common example is when calling a function that does not match the function's definition, such as parameters with incorrect datatype or name.
Generated code	result = **pd.DataFrame(df, columns=df.columns, suffixes=('_d', '_z'))**
AttributeError	This occurs when trying to access an attribute that does not belong to the class, module or submodule.
Generated code	C = tf.tensordot(A, B, axes=[[2], [2]]) sess = **tf.Session()** print(sess.run(C))
ImportError	This happens when trying to solve a built-in class or function from a library, and it is not a member.
Generated code	**from sklearn.externals import joblib** joblib.dump(fitted_model, 'sklearn_model.pkl')
ModuleNotFoundError	This module does not exist in the environment and/or in real life.
Generated code	**import numpy_indexed as npi** result = npi.group_by(accmap).sum(a)

Table 6: Examples and descriptions of type of bugs considered as imaginary features.

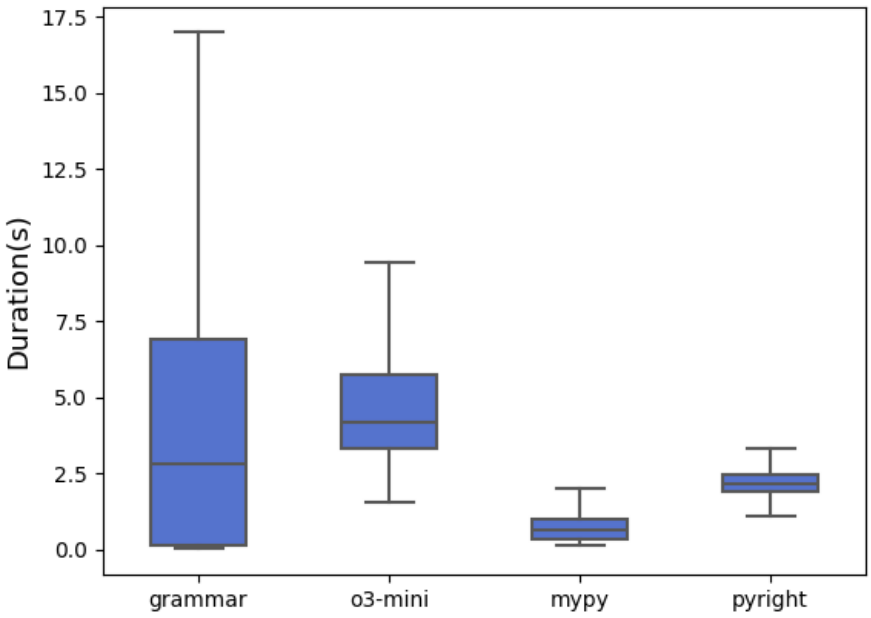

Figure 3: Comparison of time taken on bug detection analysis tools.

Static Analyzers

Can do the same as in the grammar (**Syntax + Library**)
Can keep data and control flow (**Flow**)
`cont = CountVectorizer()`
is of type CountVectorizer, and should have N attributes.
Should keep track of (**Lambda**)
Can know global and local scopes (**Scope**)
`def random(char)` vs `from random import random`
`def shift(something)` vs `shift =`

Grammars
Find syntax errors (**Syntax**) Closing parenthesis. Variable names cannot start with digits. Something needs to be at the right of = Functions and if statements should not be empty and must be followed by a statement block. Docstring grammar (**Library**) Defines the function in a library Defines a parameter in the function Defines a constant in the library
Infeasible
TEST: the tools don't know the test cases of the benchmark, so an error message generated from a test case is not handled by these tools. **AMBIGUOUS**: the tool needs to know the input and output descriptions of datatypes in the prompt to generate code accordingly. **LOGIC-DATA**: the tool cannot know if filtering a dataframe will make the series empty, it just knows it is a dataframe. Or if the LLM output is too vague or has no code ex. `END SOLUTION`.

Table 7: Examples and descriptions of annotation categories for the capability of tools to detect the error, and the source of error.

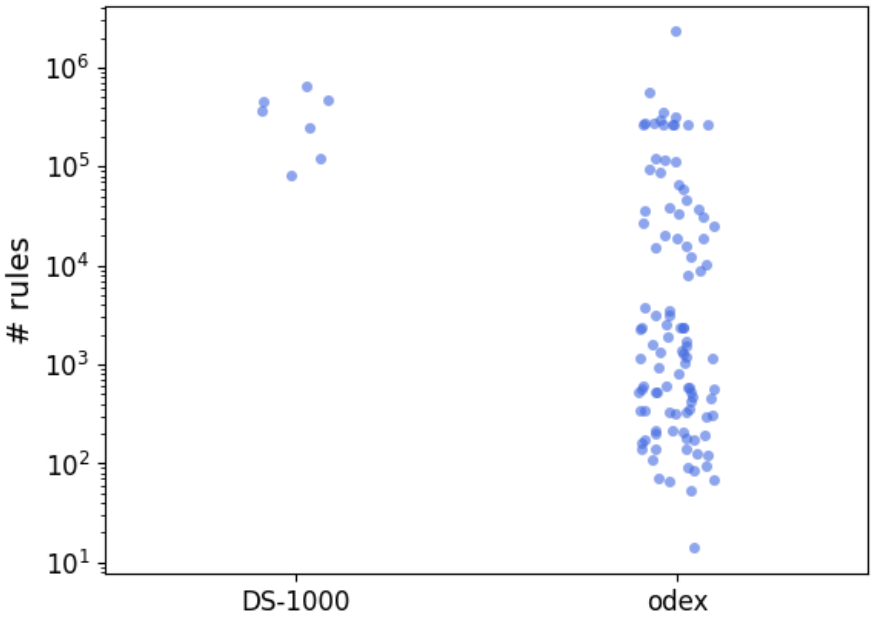

Figure 4: Distribution of the number of rules in the grammar for each benchmark.

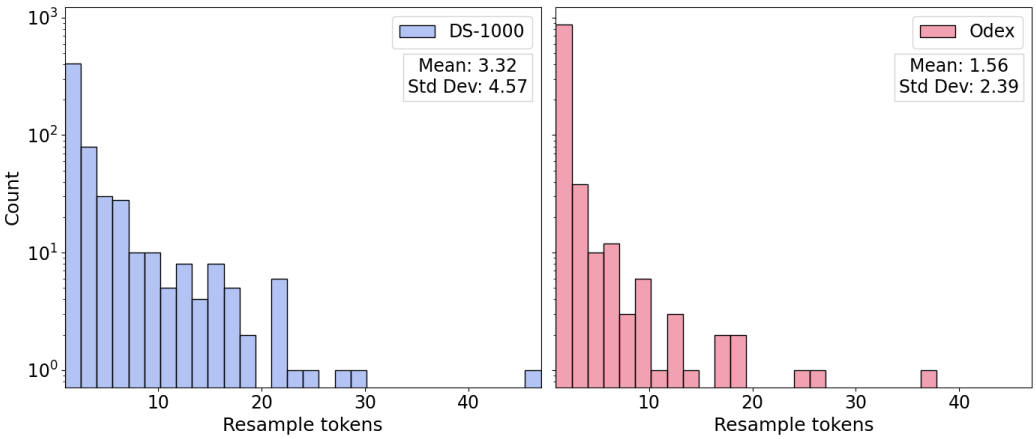

Figure 5: Distribution on resampling in each benchmark.

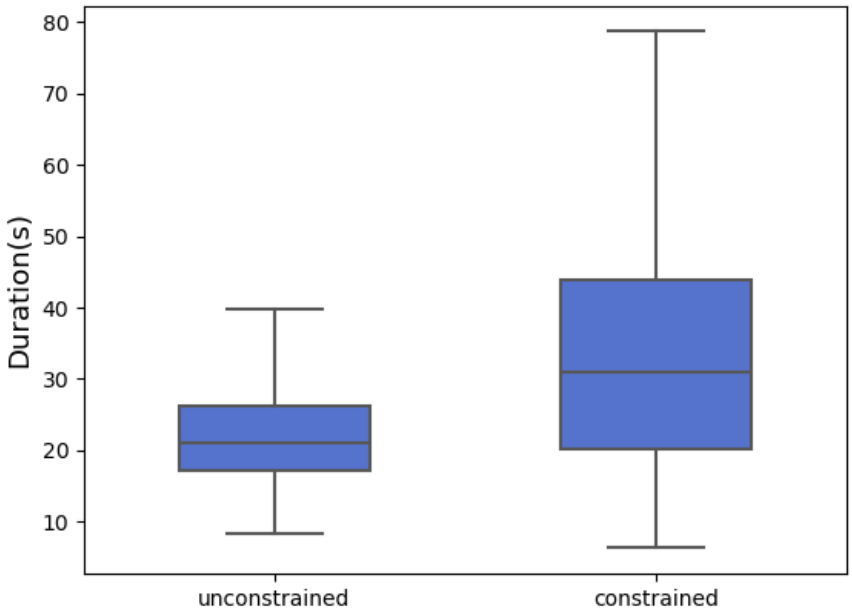

Figure 6: Comparison of time taken between unconstrained and constrained.

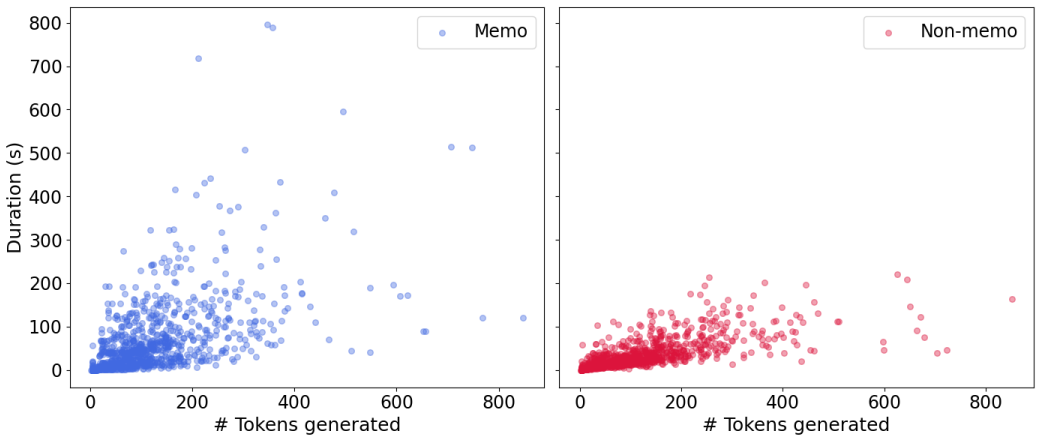

Figure 7: Comparison of time taken for approaches on grammar-constrained parsing.