Automated Grammar Generation Using LLMs

Aman Ganapathy Manvattira Annaelle Baiget Arthur Lu

Emmett Cocke Krish Patel James Shiffer

Abstract

Large language models (LLMs) have a high potential for processing highly structured text inputs to generate grammar representations. Leveraging LLMs in generating grammar would reduce the time spent and effort required to create a grammar for fuzzing, unit testing, and input validation. In this project, we create a system that handles grammar creation using automated feedback and human feedback. We develop a pipeline for assisted generation of grammar on unseen domains. We show the potential for LLMs to generate complex grammars which can be used for many software testing applications and reflect on its limitations with complex unseen domains.

1 Introduction

Generating grammar can be a complex and time-consuming task. For complex domains, grammars may have many interacting rules which may be recursive. Additionally, evaluating said grammar for correctness can be taxing as well, and optimizing grammar according to a metric can be an opaque task. There is also the underlying question of what metrics are relevant.

In this paper, we attempt to use LLMs to automate the process of grammar generation. LLMs have shown impressive text generation and reasoning capabilities for downstream tasks [8]. We attempt to leverage these capabilities for grammar generation, relying on user feedback to guide said generation in the right direction. For example, letting the user tell the LLM to adjust the grammar it generated to have fewer non-terminal nodes. Additionally, we explore several different types of metrics by which grammar can be evaluated and discuss how our system responds to them.

Since the focus of this project is to explore an LLM's ability to generate grammar, we do not want it to rely on any grammar it may have seen before. If it was asked to generate a grammar for a particular domain and it replied with a grammar heavily inspired by one in its pre-trained knowledge set, this would not be evaluating its reasoning capabilities. Consequently, we choose 3 domains so as to avoid this problem.

1.1 Choice of Domain

1.1.1 HTTP3. HTTP/3 is the latest version of the Hypertext Transfer Protocol, an application layer network protocol to transfer packets. HTTP/2 grammars are plentiful on the internet, so it is quite possible that they can be found in the pre-trained knowledge set of the LLM. We chose this grammar for two reasons: 1) The LLM will likely struggle the least with this domain so it will serve as a useful point of reference and 2) Since it has similarities to HTTP 2, it is possible the LLM will hallucinate and generate an HTTP 2 grammar instead, which is a useful data point to study

1.1.2 Nginx. Nginx is an open-source web server and proxy. Working with nginx involves constructing a well-formed configuration file describing the server's properties and routes. This strict format requires specific constructions for each "directive" and its parameters. For example, each directive can be a simple directive or a block directive, each with a different format. Additionally, each directive can have no parameters, a specific number of parameters, or a flexible list of parameters depending on each directive. Each parameter may be one of many types such as Unix file paths, URIs, boolean values, etc. Although there are many example configurations on the internet, no grammar exists to parse them, and we expect that the complexity of the domain will be a good challenge for the LLM.

1.1.3 Brewin. Brewin is a custom language created by James Shiffer for UCLA CS 131. It is an interpreted language similar to LISP. We chose this domain because it is unlikely that the LLM has seen code examples of Brewin or any parser code or trees which might inform its creation of a grammar. We expect Brewin to be a difficult challenge for the LLM since it cannot just search its knowledge for the answer and must synthesize a grammar from scratch.

2 Related Work

Few-shot prompting is a type of in context learning where examples of the task to be accomplished are provided to the LLM in the prompt. It has been demonstrated that few shot prompting improves LLM performance on downstream tasks [1]. Consequently, we felt it would improve our system's grammar generation capabilities.

Many existing tools have been developed using grammar for automating software development tasks like ANTLR and grammarinator. ANTLR defines a context free grammar format which can be parsed with the tool to generate the lexer, parser, and abstract syntax tree from the grammar [6]. ANTLR grammar can be used in grammarinator to generate random test cases through fuzzing [3].

3 Implementation

We implement our pipeline with a Gradio UI frontend, langchain middleware, and backend metrics using ANTLR and domain-specific oracles. In our pipeline, the user inputs their request into the model using the Gradio UI, which calls the langchain middleware. The middleware calls the GPT4 API and then calls the backend evaluation. After receiving the evaluations, the results are returned to the user through the frontend with the generated grammar.

3.1 LLM Implementation

We set up a prompt template that tells the LLM to treat user queries as requests for grammar. This template also takes in the history of the conversation and the metrics as input. We take advantage of few-shot prompting to teach the LLM how to format the response such that it is parseable by grammarinator and our metric evaluators. We also add several instructions to teach the LLM how to reason from the prompt what kind of grammar it should generate and what common pitfalls to avoid

3.2 Retry Budget

Since LLMs are prone to hallucinations where they can deviate from the user instructions [4], it is possible for the generated output to sometimes not be in a grammarinator parsable format. This means the rest of the system will fail to work with it, and thus we must ensure our system is robust to such hallucinations. We establish a "retry" mechanism, where if an exception is raised during any stage of the evaluation process, we prompt the LLM again for an answer. We "retry" up to a certain limit, and these retries are hidden from the frontend.

3.3 Metric Implementation

To quantify the quality of generated grammars we needed to create a mechanism which, given a textual response from an LLM, can compute objective metrics. We implemented these mechanisms using ANTLR. We use ANTLR to generate a generic visitor by providing it with a valid grammar of the ANTLR grammar creating a generic visitor which can be thought of as a meta-grammar visitor. We then overrode visitor methods so when our meta-grammar visitor is used to parse the response of an LLM we can do computations based on the nodes in the grammar being evaluated.

To generate the structure metrics we wrote another algorithm that parses the grammar file in the g4 format and generates a call graph accordingly using the library dot. We then use the call graph to compute the structural metrics. We also render the graph as a temporary PNG file to display it in the Gradio UI as output so that the user can have a better representation of the grammar generated by the LLM. The user can use this call graph to give better feedback to the LLM.

3.4 Grammarinator

We use grammarinator to return feedback on grammar validity. We call grammarinator to compile the grammar and check for syntax, structure, or other errors in the grammar. If the grammar does not successfully compile, the errors from grammarinator are returned to the LLM, which can use the feedback to fix its grammar.

3.5 Domain Metrics

Domain metrics were called to provide metrics and feedback to the user in online mode. For each domain, an oracle was used to evaluate the grammar. We return any errors in that process to the user, who can use the feedback to guide the LLM towards better grammar. The quantitative feedback from the domain oracle is also automatically fed back to the LLM.

4 Evaluation

We based our metrics analysis on *On Defining Quality Based Grammar Metrics* [2] and *A metrics suite for grammar-based software* [7]. Those documents provided the size and structural metrics that we used to assess the complexity and maintainability of the grammar generated by the LLM. We also generated metrics on three domains: HTTP/3 headers, nginx configuration files, and Brewin. HTTP/3 is a relatively new protocol but is similar to HTTP/2 so we expect the model to perform well on this domain. We selected nginx because there are few publicly available grammars for parsing nginx configuration files, so we believe that it will be a sufficient challenge for the model. Finally, Brewin is a custom language created for CS 131 at UCLA so we believe no LLM has ever seen examples of this domain before and will be a hard challenge.

4.1 Size Metrics

We implemented five metrics that provide insights into the size and complexity of the grammar. We used ANTLR to generate a visitor for the grammar file generated by the LLM and compute the size metrics accordingly.

4.1.1 Number of terminals (TERM). Measures the total number of terminal nodes in the grammar. A high value indicates that the language has a rich vocabulary which would make parsing more challenging and lexer rules more extensive. However, a low value suggests a simpler language which might be easier to process but potentially less expressive.

4.1.2 Number of non-terminals (VAR). Measures the total number of non-terminal nodes in the grammar. A large number of non-terminals implies a greater maintenance overhead since changes to the definition of one may affect many others.

4.1.3 McCabe Cyclomatic Complexity (MCC). Measures the number of independent paths through a flow graph. A higher MCC value suggests a more intricate structure with significant recursion and dependencies among rules. This indicates a powerful but potentially harder-to-understand grammar. On the other hand, a lower value would suggest a simpler grammar with fewer interdependency.

4.1.4 Average RHS size (AVS). Measures the number of symbols that we can expect to find on average on the right-hand side of a grammar rule. It gives insights on how detailed and dense the grammar is. A high value would mean that the rules are more complex and thus it could lead to higher chances of ambiguity and add more difficulty to the parsing process. On the other hand, a lower value reflects a simpler grammar.

4.1.5 Halstead effort (HAL). Measures the complexity of the grammar considering the number of operators (grammatical operations) and operands (terminals and non-terminals) weighted by grammar size. A high value means that the grammar is high in complexity due to a lot of interrelations among rules or densely packed semantics. A lower value suggests that the grammar is simpler. Compared to MCC, it provides a better basis for judging differences in complexity between grammars of different sizes.

4.2 Structural Metrics

The five structural metrics we implemented provide insights into the complexity and interdependency within a grammar. Every metric has a distinct function in assessing a grammar's structure and how it affects software maintenance. To compute these metrics, we first had to build a call graph of the grammar generated by the LLM. Then, we analyzed the call graph to determine those metrics.

4.2.1 Tree Impurity (TIMP). Measures how close the call graph is to a tree structure. To compute this metric, we do the ratio of edges to nodes, which indicates how interconnected the non-terminals nodes are within the grammar. High TIMP values suggests that the call graph is a dense network of dependencies, which can make the parsing process more complicated and potentially lead to design issues. As a result, the goal is to have the TIMP value approach 0.

4.2.2 Number of Levels (CLEV). For this metric, we use the call graph to partition the non-terminals into a set of equivalence classes called grammatical levels. CLEV is calculated as the percentage of the actual number of levels relative to the maximum possible number of levels. A low CLEV value suggests that the non-terminals nodes are grouped in a few equivalence classes which indicates that the grammar is easier to understand, maintain, and extend. A high value of CLEV suggests that the non-terminals nodes are more evenly distributed across levels. Therefore, it might be possible to reorganize them into more equivalence classes. The goal for the model is to minimize CLEV by optimizing the grammar.

4.2.3 Non-singleton Level (NSLEV). Counts the number of equivalences classes derived by the call graph that have more than one non-terminal node in it. This metric gives an idea of how many equivalence classes define the core logic of the grammar. Indeed, [2] explains that central language concepts, such as declarations, expressions and statements tend to be represented by larger classes, highlighting logical groupings in the grammar. A high NSLEV value suggests that the grammar has various clusters of closely related rules. It could indicate a modular and reusable design. On the other hand, a low NSLEV value would indicate a language with less mutual recursion that is more fragmented or overly detailed.

4.2.4 Size of largest level (DEP). Measures the number of non-terminals in the largest grammatical level (derived from the call graph). DEP measures balance of the distribution of non-terminals nodes is across the different levels. A high DEP value, especially if it represents a large proportion of the total non-terminals, means that the distribution is uneven. This could indicate poor modularization or excessive complexity. On the other hand, a low DEP value suggests an even distribution across the different levels which could correspond more to a well-structured and modular grammar.

4.2.5 Maximum height (HEI). Measures the maximum height of the call graph, which corresponds to the longest path across the different levels. This metric gives another measure of the dispersion of the non-terminals among the grammatical levels.

4.3 Domain Metrics

For each domain, we define a positive set of examples which are valid in the domain. For example, all the valid nginx configuration files which would normally pass validation by the nginx runtime. Similarly, we can define the set of all invalid examples in the domain. We generate a small set of examples of varying complexity for each domain with positive and negative examples from each set. We expect an ideal grammar to be **complete** and **correct**.

- By **complete** we mean that the grammar parses every positive example without errors, and all of the correct relational structure is extracted.
- By **correct** we mean that the grammar rejects every negative example and identifies the error, and any incorrect relational structure is rejected.

To measure completeness and correctness, we run the model's generated grammars on the test set and generate a confusion matrix. We make the confusion matrix of true positives (positive examples correctly parsed), false positives (negative examples incorrectly parsed), true negatives (negative examples correctly rejected), false negatives (positive examples incorrectly rejected). From the confusion matrix, we generate the F_1 score defined in equation 1.

$$F_1 = \frac{2 * TP}{2 * TP + FP + FN} \tag{1}$$

We evaluate the best grammar generated by the model on each domain using the F_1 score. This score incentivize the model to maximize true positives (TP) while minimizing false positives (FP) and false negatives (FN). The best score is 1.0 and the worst score is 0.0.

4.3.1 HTTP/3. HTTP requests are web requests that contain a start line, headers, and body. The start line defines the method and HTTP version, the headers are specifications about the request, and the body contains the data payload. Due to the ubiquity of HTTP, we anticipate that the LLM will effectively generate correct grammar since its training corpus has many instances of valid HTTP requests. Specifically, we believe that it will generalize trivially to HTTP/3, despite it being a recent version, since the only required change compared to past versions will be changing the version number in the start line. Therefore, we feel this serves as a good baseline of the LLM's capabilities in generating desired grammars for our other specific domains.

We assess the validity of generated grammar by having the URL for fuzzed examples be "https:// postman-echo.com", a website tailored for diagnosing HTTP request correctness. We measure the performance of 25 fuzzed examples from the grammar with 5 perturbed examples to evaluate the confusion matrix.

Our findings indicate that the grammar correctly parses validly-formed HTTP requests at a 100% frequency on the test suite. However, it also validly parses all of the perturbed tests as well. This is because the perturbations are applied to the URL. Since there is a theoretically infinite number of valid URL's, the grammar is incapable of identifying incorrect URL's since that can only be done empirically. Thus, this permittivity is accounted for. We opted not to apply other perturbations since we have found postman-echo to be permissive for malformed header and body fields. Thus, modifying these does not provide additional insights since the ground truth for them will be that they are well-formed and so the confusion matrix will remain composed of only true-positives.

4.3.2 Nginx. Nginx configuration files are highly structured files with a limited set of "directives" each with specific semantic requirements. For example, each directive may have multiple parameters, each of different types. Additionally, each directive has a specific scope which limits its relation to other directives. We believe that this domain will be challenging for an LLM to generate complete and correct grammar. We also write a baseline grammar for this domain to compare against the LLM generated grammar and can be found in Appendix B.

We evaluate the LLM generated grammar against a simple baseline (appendix B), which is complete but not correct. We evaluate based on a set of 5 correct and 5 invalid example nginx configuration files and generate the confusion matrix. The grammar is used to parse each example nginx configuration file. For positive examples, we expect the grammar to successfully parse the example without errors. For negative examples, we expect the grammar to return an error and unsuccessfully parse the example. We generate a confusion matrix for the grammar.

4.3.3 Brewin. Brewin has standard features of any strongly typed object-oriented language, such as functions, classes, and exceptions. Syntactically, it is not unlike Lisp, with all expressions being enclosed in parentheses, sometimes with deep nesting, and operators ordered in Polish notation. Few-shot prompting with code examples is used as a basis for grammar generation.

The LLM-generated grammars were tested on a set of 17 reference Brewin programs, 12 of which were known to run without issues, and 5 of which returned errors from the canonical interpreter, ranging from syntax errors to incorrect keywords to duplicate definitions. Similar to Nginx, a confusion matrix was generated, and used in computing the F1 score metric.

5 Results

We run the evaluation metrics on each domain and record the best grammar generated over several iterations of human feedback. We compile a table for each domain showing the metrics for the best grammar.

5.1 HTTP/3 Online

F1: 0.9 (validation set: 25 positive, 5 negative)

TERM	24	TIMP	4.58
VAR	17	CLEV	1
MCC	4	NSLEV	3
AVS	2.9	DEP	3
HAL	778	HEI	6

Table 1: HTTP/3 Online Metrics Size & Structural

5.2 Nginx Online

F1: 0.0 (validation set: 5 positive, 5 negative)
Baseline F1: 0.666

TERM	94	TIMP	0.392
VAR	31	CLEV	1
MCC	40	NSLEV	4
AVS	5.84	DEP	19
HAL	5979	HEI	4

Table 2: Nginx Online Metrics Size & Structural

5.3 Brewin Online

F1: 0.15 (validation set: 12 positive, 5 negative)

TERM	26	TIMP	10.5
VAR	15	CLEV	1
MCC	13	NSLEV	3
AVS	5.33	DEP	5
HAL	2395	HEI	8

Table 3: Brewin Online Metrics Size & Structural

5.4 Brewin Offline Graphs

We run the model on the Brewin domain for 10 iterations without human feedback as "offline learning". We measure the metrics on each iteration and graph the metrics over time to show how well the LLM is guided by the metrics alone on the least seen domain.

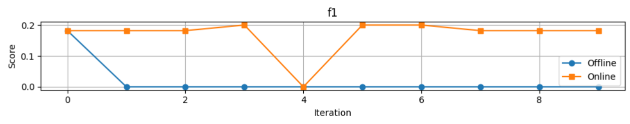

Figure 1: Brewin Offline (10 iters) F1 Score

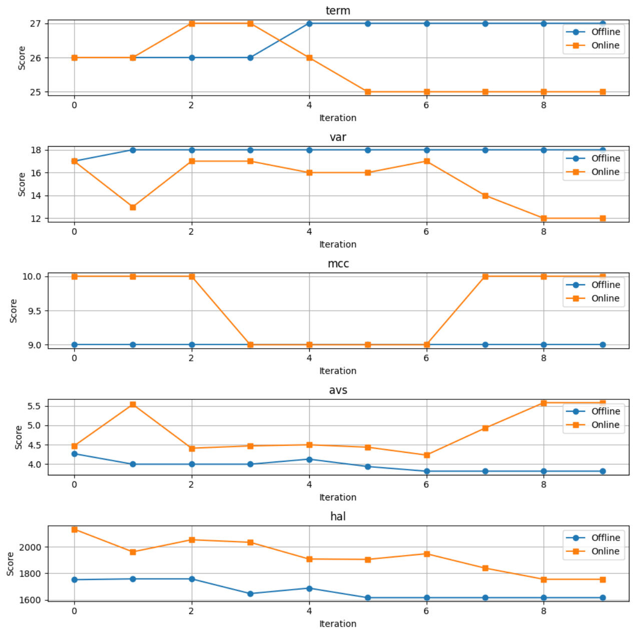

Figure 2: Brewin Offline (10 iters) Size Metrics

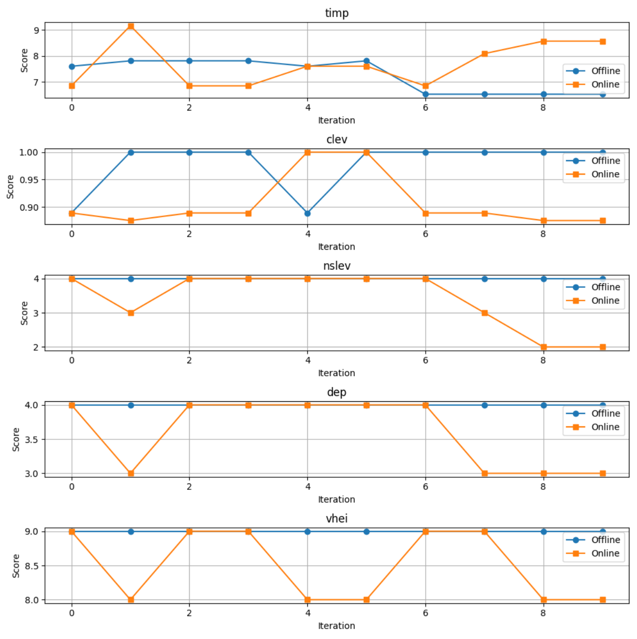

Figure 3: Brewin Offline (10 iters) Structure Metrics

Fig 2 and Fig 3 highlight key differences between the online and offline grammar generation approaches. The AVS metric exhibited a downward trend in both modes until online intervention removed a non-terminal, increasing AVS but improving other metrics overall. HAL consistently decreased across both modes, reflecting steady optimization. Most importantly, the F1 score shown in Fig 1, which dropped to zero at some point in both modes, failed to recover offline, whereas online intervention allowed the LLM to address errors and improve. Interestingly, seven metrics improved in online learning versus only three offline over the course of 10 iterations, suggesting there is value in human guidance. The increased "jittering" in online metrics further demonstrates the stronger influence of human intervention enabling better grammar generation.

5.5 Generated Nginx Grammar Call Graphs

The nginx performance was not as expected, so we generated the grammar call graphs for the nginx grammar produced during the online generation with human feedback. During generation, we noted that the grammar continued to switch between two states shown in Fig 4 and Fig 5 (Appendix C)when the LLM was prompted to either connect all the rules or ensure that all lexer rules were defined.

6 Conclusion

In our project, we experiment with various LLM techniques such as few-shot prompting, and online and offline feedback models, towards automated generation of grammars on unseen domains. We show that LLMs have a promising role in the automated generation of grammars, but ultimately show that such models find most unseen grammar domains challenging. We note that domains that are close to existing knowledge in the LLM such as HTTP/3 performed better, and shows that the LLM can modify existing known grammars. However, our model struggled with generating new grammar on completely unseen domains.

We also attempted to incorporate Retrieval Augmented Generation on a separate branch of our code (setup-RAG-brewin) to see if it would improve the results on Brewin. We uploaded a pdf containing more information on Brewin which was broken apart into vectors and retrieved with a retriever when the LLM needed it. The hypothesis was that giving it more knowledge on the domain would help it perform better, as user prompts are an inherently limited medium and this was a completely new domain. However, this did not help it at all. There was no demonstrable difference in generation quality. This indicates that the problems with using LLMs to generate grammars are not linked to a lack of information on the domains said grammars would be used for.

Additionally, as discussed above, the two callgraphs for nginx were simply shuffled around versions of each other with issues inherent to their design that made them unsuitable. This is despite the prompting indicating what errors the LLM made and how to fix them. This indicates that the LLM lacks an implicit understanding of what it is doing. This would line up with the results demonstrated by McKenna, which show that due to a series of biases inherent in LLMs, they lack implicit reasoning capabilities [5] Consequently, when asked to generate a grammar for a domain, it lacks implicit understanding of how the different parts of the grammar connect. This indicates one direction to study is using LLMs to generate parts of grammar rather than the entire grammar from top to bottom and gluing them together.

Future experimentation could focus on improving the quality of grammar generated by the LLM. We suggest a few approaches based on our findings that may improve a model's performance.

- We believe that better model selection will result in better results. The GPT4 model we selected is useful as a general-purpose LLM, however on the specific task of generating grammar, a custom-trained model will likely perform significantly better. Some instruction fine-tuning may be in order.
- Better feedback interpretability may allow the model to respond better to automated feedback. Parsing grammar errors to give targeted feedback was done manually, and may not have been ideal. Automated parsing and interpretation of grammar errors could allow the model to fix grammar and improve on the metrics.
- Our testing shows that the distance between our chosen domains and the LLM's knowledge domains was a large factor in the model's performance. The HTTP/3 domain performed perhaps because of its similarity to HTTP1 which is within the model's knowledge. Better

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver,

BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[2] Julien Cervelle, Matej Črepinšek, Rémi Forax, Tomaž Kosar, Marjan Mernik, and Gilles Roussel. 2009. On defining quality based grammar metrics. In *2009 International Multiconference on Computer Science and Information Technology*. 651–658. https://doi.org/10.1109/IMCSIT.2009.5352768

[3] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3278186.3278193

[4] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2024. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Information Systems* (Nov. 2024). https://doi.org/10.1145/3703155

[5] Nick McKenna, Tianyi Li, Liang Cheng, Mohammad Hosseini, Mark Johnson, and Mark Steedman. 2023. Sources of Hallucination by Large Language Models on Inference Tasks. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 2758–2774. https://doi.org/10.18653/v1/2023.findings-emnlp.182

[6] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 425–436. https://doi.org/10.1145/1993498.1993548

[7] James F. Power and Brian A. Malloy. 2004. A metrics suite for grammar-based software. *J. Softw. Maintenance Res. Pract.* 16 (2004), 405–426. https://api.semanticscholar.org/CorpusID:8626516

[8] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2024. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL] https://arxiv.org/abs/2303.18223

A Running The App

The project git repository can be found here (link). Instructions can be found in the README. After cloning the repiository, please install the requirements by running:

```
pip install -r requirements.txt
pip install antlr4-python3-runtime==4.13.0
pip install antlr4-tools==0.2.1
pip install graphviz
sudo apt get graphviz
```

You will also need to install thge java runtime and nginx. The app can be run from frontend/app.py.

B Nginx Baseline Grammar

```
grammar baseline_nginx;

// Parser rules
config: directive+ ;
directive: block | simple ;
block: blockDirective '{' directive* '}' ;
blockDirective: name | name params;
simple: simpleDirective  params';' ;
simpleDirective: name;
name: STRING ;
params: param (param)* ;
param:
    NUMBER
  | STRING
  | VARIABLE
  | URL
;

// Lexer rules
NUMBER: [0-9]+ ;
STRING: [a-zA-Z0-9._/|\\=^*$()[\]'":~-]+ ;
VARIABLE: '$' [a-zA-Z_][a-zA-Z0-9_]* ;
URL: 'http://' [a-zA-Z0-9.-]+ ;
COMMENT: '#' [ -~]+ -> skip ;
WS: [ \t\r\n]+ -> skip ; // Skip whitespace
```

C Generated Nginx Grammar Call Graphs

Generated call graphs for the nginx grammar can be found on the following pages.

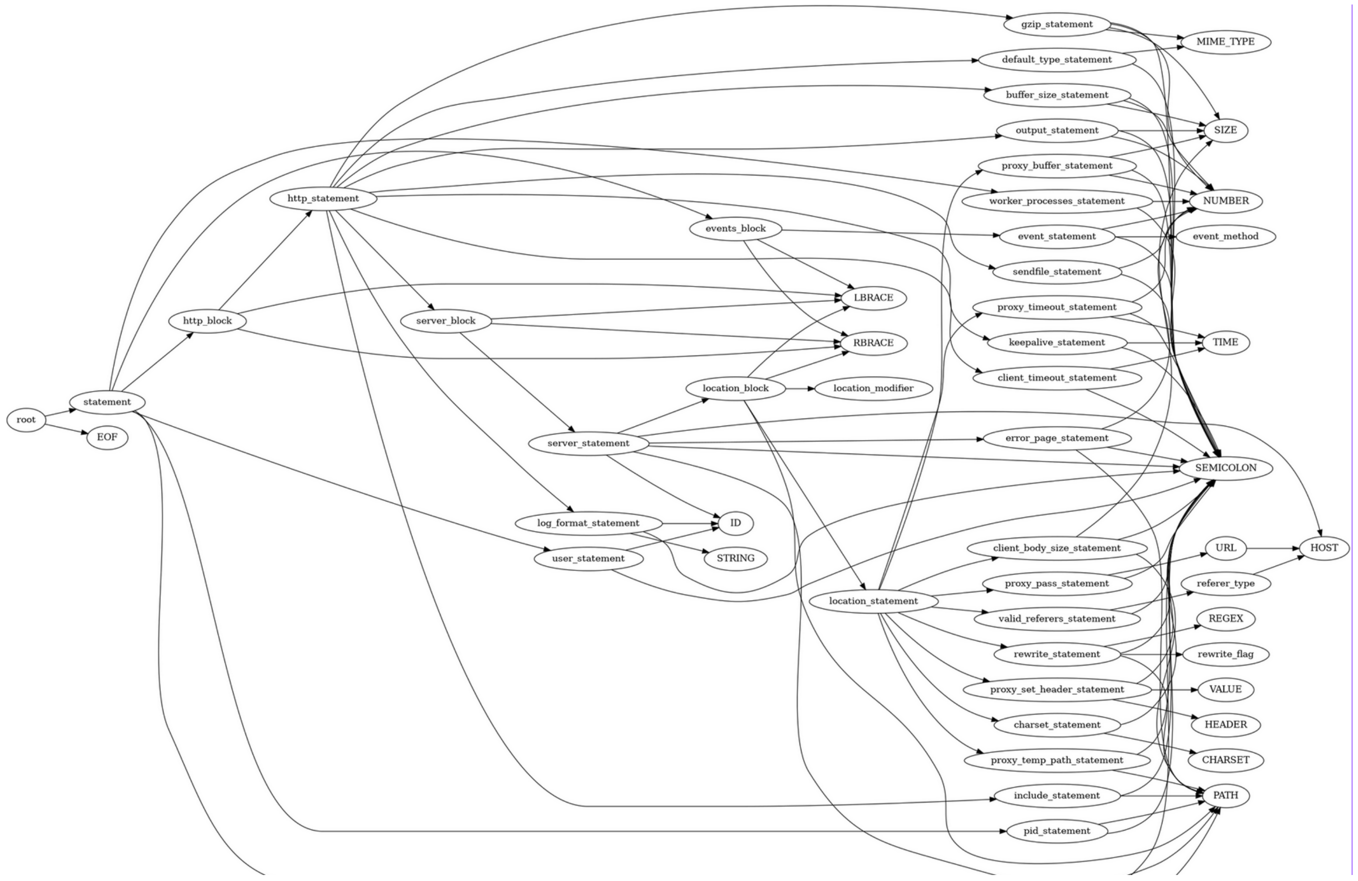

Figure 4: Nginx Call Graph 1

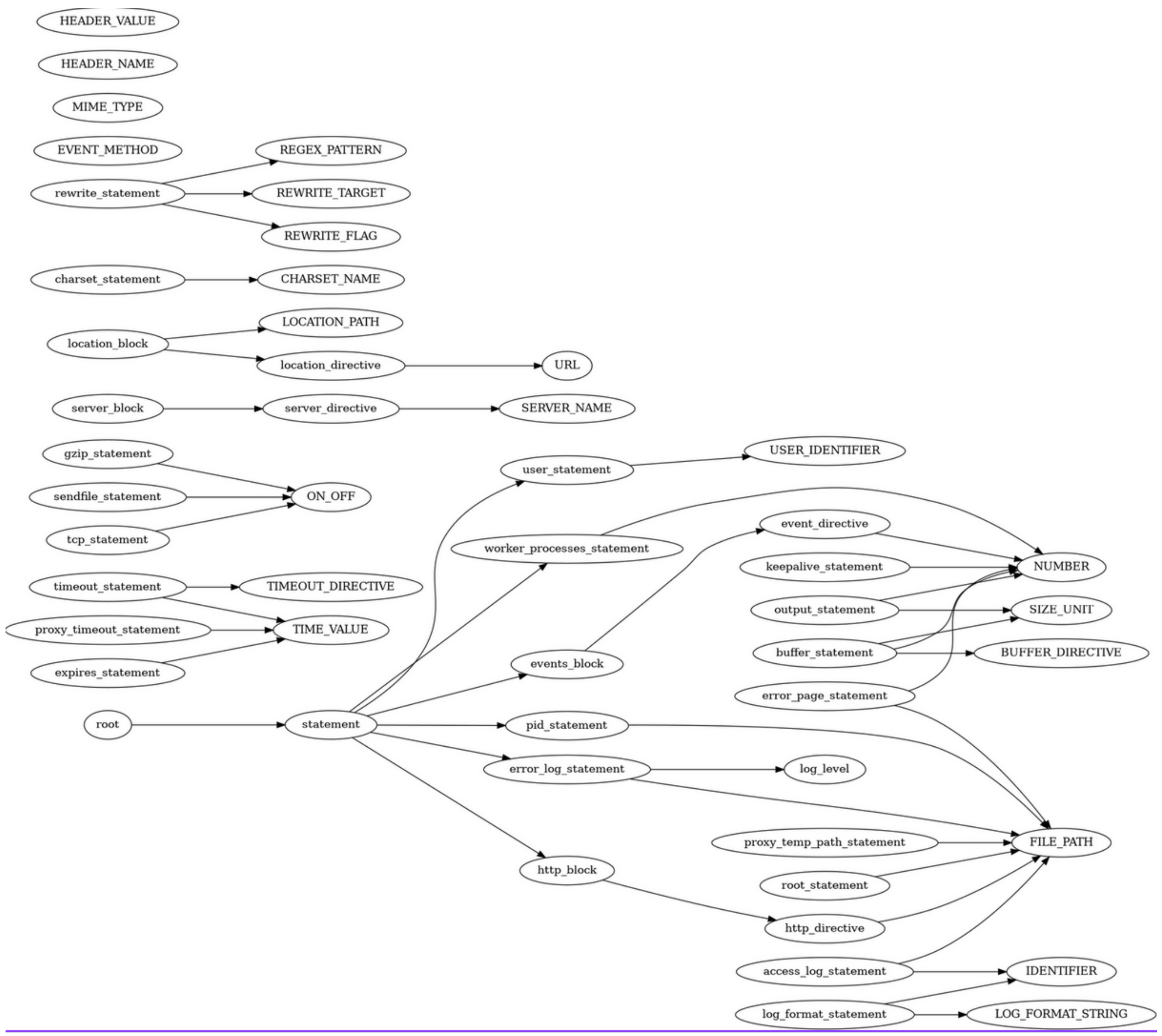

Figure 5: Nginx Call Graph 2