Using a Feedback Loop for LLM-based Infrastructure as Code Generation

Mayur Amarnath Palavalli
mayur.palavalli@gmail.com
Irvington High School
Fremont, California, USA

Mark Santolucito
msantolu@barnard.edu
Barnard College, Columbia University
New York, New York, USA

Abstract

Code generation with Large Language Models (LLMs) has helped to increase software developer productivity in coding tasks, but has yet to have significant impact on the tasks of software developers that surround this code. In particular, the challenge of infrastructure management remains an open question. We investigate the ability of an LLM agent to construct infrastructure using the Infrastructure as Code (IaC) paradigm. We particularly investigate the use of a feedback loop that returns errors and warnings on the generated IaC to allow the LLM agent to improve the code. We find that, for each iteration of the loop, its effectiveness decreases exponentially until it plateaus at a certain point and becomes ineffective.

ACM Reference Format:

Mayur Amarnath Palavalli and Mark Santolucito. 2018. Using a Feedback Loop for LLM-based Infrastructure as Code Generation. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Infrastructure as Code (IaC) has fundamentally transformed the way cloud infrastructure is managed. With IaC, developers can provision and maintain their infrastructure through code. This ensures automation and consistency in deploying infrastructure, allowing for more effective scaling of operations. IaC also allows teams to use version control on their infrastructure, making it easier to collaborate and track changes [13]. However, a major challenge with IaC is in the difficulty of writing correct code [10, 12, 15, 16, 18].

At the same time, Large Language Models (LLMs), as applied to code generation, are enabling developers to be more effective. Code generation benchmarks, such as HumanEval [11] and SWEBench [14], have shown that LLMs are capable of assisting developers with challenging programming tasks. It is then natural to seek to extend the application of LLMs for code generation to IaC.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

The combination of IaC and LLMs could allow for wider adoption of IaC and more effective infrastructure management. However, many questions remain open about the ability of LLMs to reason about the complexities of IaC. In particular, there are many implicit rules about the semantics of cloud resources that are difficult to reconcile when creating infrastructure through IaC.

In our work, we provide an analysis of the ability of LLMs to generate AWS CloudFormation code. In particular, we investigate the ability of an LLM to respond to errors and warnings generated by `cfn-lint` [3], a tool for analyzing CloudFormation code. In summary, the main contributions of this work are:

(1) The design of a feedback loop system for the generation of CloudFormation code.
(2) A set of CloudFormation code generation benchmarks built from industry-standard IaC problems.
(3) An evaluation of our feedback loop system on this benchmark, showing that LLMs struggle to fully reconcile all errors in IaC generation.

2 Related Work

The application of LLMs in generating IaC has amassed significant attention in recent years. Using LLMs for IaC comes with certain challenges that make it inefficient in large systems. Srivatsa et al. evaluate the LLM performance on functional correctness by comparing with human-written code and deciding whether or not it is an exact match. They found that the GPT-3.5-turbo model had a success rate of between 50 and 60 percent, while the Codeparrot model never exceeded 10 percent accuracy [19]. They also discuss ethical and safety concerns of developing LLMs for more accurate IaC generation.

A natural extension is the work of Ugare et al. who introduce SynCode [21], a framework that uses grammar rules to enhance LLM generation in formal coding languages. SynCode is able to reduce 96.07 percent of syntax errors in Python and Go. It particularly shines in generating JSON, where it is able to eliminate all syntax errors. This is achieved by utilizing context-free grammar rules based on discrete finite automation.

An important area of investigation related to LLM code generation is constrained decoding [7, 17]. At a high level, grammar constrained decoding (GCD) helps language models (LLMs) in producing structured results without requiring additional fine tuning. GCD utilizes grammars to guarantee that the generated sequences follow a predefined structure. The technique greatly improves the performance of LLMs in such settings without the need for expensive task-specific training efforts. Since we are generating highly structured JSON documents (CloudFormation files), GCD could be a way to further optimize the results. However, the main goal of our work is to provide an initial baseline accounting of the viability of LLM code generation for IaC. We leave such optimizations to future work.

Another critical challenge in LLM code generation is fixing syntax errors. Tsai et al. address this with RTLFixer [20], a framework designed to fix syntax errors in Verilog code. After finding that 55 percent of errors in LLM-generated Verilog code were syntax errors, RTLFixer was designed to utilize Retrieval-Augmented Generation and ReAct (Reasoning and Action framework) techniques to improve error correction. The framework achieves a 98.5 percent success rate in fixing syntax errors after testing on 212 syntactically invalid Verilog implementations.

These studies demonstrate the evolving role of LLMs in code generation and highlight the importance of addressing syntactic correctness and complexity.

3 Background

```
{"AWSTemplateFormatVersion": "2010-09-09",
"Resources": {
  "MyEC2Instance": {
    "Type": "AWS::EC2::Instance",
    "Properties": {
      "InstanceType": "t2.micro",
      ...
```

Figure 1: An example AWS CloudFormation JSON template.

IaC is a diverse space with many existing languages and tools. The most widely adopted tools are Terraform [5] and Amazon Web Service's CloudFormation [9]. Both Terraform and CloudFormation files are specified as JSON (or YAML) documents, where the various fields define properties of the cloud resource to be deployed. These files are generally declarative - giving a specification of the desired cloud infrastructure state. This is in contrast to some other IaC languages, such as Pulumi, which allow users to write imperative programs that construct operations that should take place upon the user's infrastructure [1]. An example of a CloudFormation JSON is given in Fig. 1.

```
E1015 {'Fn::GetAZs': ''} is not of type 'string'
Error location - path/to/my_iac.json:1:4575
```

Figure 2: An example error message from `cfn-lint`.

In this work, we focus solely on the LLM generation of AWS CloudFormation [9] in JSON. We additionally use the AWS CloudFormation linter `cfn-lint` [4]. `cfn-lint` allows us to validate CloudFormation JSON templates against the resource provider schemas provided by AWS as well as other best-practice IaC rules. A schematically valid JSON template returns nothing when run through `cfn-lint`. In any other case, `cfn-lint` returns errors and/or warnings. The example error in **Fig. 2** outlines the basic structure of an error produced by `cfn-lint`. The error message code starts with a letter followed by a string of numbers which forms a error code. This is followed by a brief description of the error and the line and character number at which it occurs.

4 Methodology

To describe our system, we first describe the methodology we used for collecting a benchmark set, then we describe the feedback loop that we constructed with `cfn-lint`. We make all of our code and evaluation results available open-source: https://github.com/Mayur-Palavalli/LLM-IaC-generation.

```
Create a AWS CloudFormation template that deploys a VPC
with a pair of private subnets spread across two
Availabilty Zones. It deploys a VPC Endpoint for
CloudFormation so an instance in the private subnet can
use cfn-signal for its CreationPolicy.
```

Figure 3: An example prompt from the official AWS CloudFormation Template Schema repository.

4.1 Benchmark Set

To create a dataset of prompts, we took 33 descriptions of AWS CloudFormation problems from the official AWS CloudFormation Template Schema repository [2]. This repository converts existing Resource Specifications files into a formatted JSON (or YAML) schema document which can be integrated in an IDE. An example prompt from this repository is displayed in Fig. 3

4.2 Feeback Loop

We queried an LLM for a solution to this problem five times for each description, yielding a dataset of 165 implemented CloudFormation templates. We generated these templates in JSON format.

The tool `cfn-lint` is able to process a JSON CloudFormation template, and returns errors describing schema violations, invalid resource properties, and best practices. After each generation, we run `cfn-lint`, and send the error message(s) and JSON template back to the LLM, instructing it to modify the template to fix the

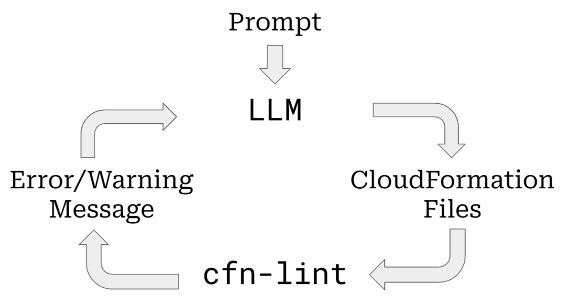

Figure 4: A diagram of the feedback loop: we provide the LLM with a prompt for an AWS CloudFormation file, which is run through `cfn-lint` to produce error/warning message(s) which are given back to the LLM.

error generated by `cfn-lint`. In this manner, we create the feedback loop outlined in **Fig. 4** that sends each JSON CloudFormation file in our dataset to the LLM, each time providing the new file and corresponding error message(s). This process is repeated ten times. We hypothesized that the LLM would be able to fix all errors produced by `cfn-lint` after a certain number of iterations.

We keep track of the total number of errors and warnings across all 165 files after each iteration of the feedback loop (for all 10 iterations) in order to identify the point at which it becomes ineffective. This process of running each file ten times through the feedback loop is repeated six times. We choose to repeat it six times because most of the error bars are narrow, indicating a small standard deviation in the number of errors after each of the ten iterations. A small standard deviation implies low variance, which demonstrates that additional repetitions are unlikely to change the overall pattern.

5 Results

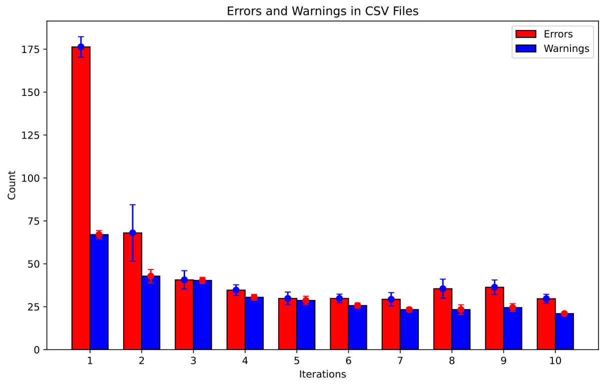

Figure 5: A histogram of errors over multiple `cfn-lint` feedback iterations showing error bars representing the distribution over six trials.

The entire process of iterating each file ten times through the feedback loop is repeated six times. **Fig. 5** summarizes the results of all six sets of iterations in a bar graph by displaying the total number of errors and warnings in all 165 JSON files after each iteration. The graph's plateau beginning at approximately the fifth iteration indicates the point at which the feedback loop is no longer effective.

The plateau is caused by the feedback loop's inconsistency in fixing errors. The LLM is unable to correctly fix certain individual errors, which occasionally result in several new errors being produced from each old error between iterations. After approximately five iterations, this anomaly is enough to yield no significant net change in the number of errors across all 165 files. The slight peak in the number of errors at the eighth and ninth iterations indicates at least one of two things:

(1) An unusually high number of files generated more errors than it had after the seventh iteration.

(2) Certain files had an unusually high increase in the number of errors generated after the seventh iteration.

We predict the cause of this to be the LLMs incapability to properly understand all the error messages from `cfn-lint`.

6 Discussion

Perfecting the use of LLMs in generating valid AWS CloudFormation could enable developers to more quickly build the infrastructure they need for their systems. Automating the generation of CloudFormation templates could drastically increases the speed and efficiency of setting up complex cloud environments.

We do not use ChatGPT's structured output mode [8]. Structured output mode allows users to provide the desired schema of JSON output from ChatGPT in addition to the prompt. This mode guarantees that the generated response matches the schema. However, for the purposes of CloudFormation generation this is not a viable option. Not only is the complete CloudFormation schema is over 200,000 lines long, it uses features of JSON schemas that are outside the scope of ChatGPT's structured output mode [8].

While our feedback loop dramatically increases the chance of generating valid IaC, we still do not have a formal guarantee that the generated code will be schematically valid. The remaining uncertainty is enough for LLM code generation of IaC to remain hard to use in large-scale development. There is a further question of semantic validity. Semantic validity captures the idea that not only is the CloudFormation deployable, but is also what the user wants (e.g. an empty file is always schematically valid, but not semantically valid). A schematically valid CloudFormation is not guaranteed to be semantically valid. We are left with the task of ensuring schematically valid CloudFormation are also semantically valid before LLMs are safe to use at large scale. This, however, is a challenging problem. Measuring semantic validity requires a way to determine how well the generated infrastructure adheres to the prompt. There is currently no simple tool like `cfn-lint` that does this.

6.1 Threats to Validity

One threat to the generalizability of our work is the question of the extent to which we would see the same pattern if using a different LLM. At the time of writing, OpenAI's gpt-4o is among the best LLMs capable of working with JSON. We believe that the use of a different LLM for our work would the rate of decrease in the number of errors, but not change the overall pattern. We hypothesize that the plateau on the graph in Fig. 5 is due to the LLM's inability to reconcile error messages with the high-level intent of the prompt. If this is true, we would need a structurally different LLM than OpenAI's gpt-4o model.

Another threat to generalizability is the extent to which the IaC provider impacts the pattern of our results. AWS CloudFormation is one of the most well-documented IaC services, and thus has large corpus of training data related to it, making it well-suited to generation with LLMs. Other well-documented IaC services, such as Terraform, would likely yield similar results due to similar compatibility with LLMs. We hypothesize that using a less documented services would likely still produce a plateau, but would yield more slowly declining error bars due to a smaller training set of relevant data.

Furthermore, our work utilizes only one method of receiving feedback. We use error messages from `cfn-lint` as feedback to demonstrate a proof of concept, but other means of testing and receiving feedback could yield different results. The plateau, for instance, might not exist with a more structured feedback strategy.

6.2 Future Work

We find that an LLM has a limited ability to respond to error messages and correct code in the context of IaC configurations. A future task would be to customize error messages to a format that an LLM like ChatGPT more clearly understands so errors can be be fixed without creating new ones. The creation of a tool like cfn-lint to check for semantic validity would take IaC generation via LLMs to the next level, allowing for widespread commercial use.

In the context of IaC, Pulumi [1] is a an open-source IaC tool that allows developers to define, deploy, and manage cloud infrastructure in familiar programming languages. All AWS services, including CloudFormation, are fully supported by Pulumi. An especially useful tool is Pulumi AI [6], an experimental feature that allows developers to generate IaC in familiar programming languages via natural-language prompts. As far as we know, the Pulumi AI tool does not incorporate any feedback during its generation from static analysis tools such as `cfn-lint`. A future task might be to use Pulumi AI to generate Pulumi code which maps to AWS CloudFormation, which is run through `cfn-lint`, and send the generated errors back to correct the Pulumi code.

7 Conclusion

Our investigation of the use of feedback loop for Infrastructure as Code generation demonstrates the potential of agentic LLM systems for real-world software development. Our results indicate that using LLMs to generate IaC provides a significant advantage, but not yet enough to be make it scale-able. The feedback loop offers a method to increase the rate of successful IaC generation from LLMs, as well as an opportunity for further research to fix the anomalies that cause the loop to become eventually ineffective. Even if OpenAI's structured output mode were to support large complex schema like that of AWS CloudFormation, there are still issues of semantic correctness that are not captured in the schema and only appear through checks with tools like `cfn-lint`.

References

[1] Documentation — pulumi.com. https://www.pulumi.com/docs/, [Accessed 12-09-2024]
[2] GitHub - aws-cloudformation/aws-cloudformation-templates: A collection of useful CloudFormation templates — github.com. https://github.com/aws-cloudformation/aws-cloudformation-templates, [Accessed 02-09-2024]
[3] GitHub - aws-cloudformation/cfn-lint: CloudFormation Linter — github.com. https://github.com/aws-cloudformation/cfn-lint?tab=readme-ov-file, [Accessed 12-09-2024]
[4] GitHub - gofireflyio/aiac: Artificial Intelligence Infrastructure-as-Code Generator. — github.com. https://github.com/gofireflyio/aiac, [Accessed 02-09-2024]
[5] GitHub - hashicorp/terraform: Terraform enables you to safely and predictably create, change, and improve infrastructure. It is a source-available tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned. — github.com. https://github.com/hashicorp/terraform, [Accessed 30-09-2024]
[6] GitHub - pulumi/pulumi-ai — github.com. https://github.com/pulumi/pulumi-ai, [Accessed 12-09-2024]
[7] Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning — arxiv.org. https://arxiv.org/abs/2305.13971, [Accessed 25-10-2024]
[8] Introducing Structured Outputs in the API. https://openai.com/index/introducing-structured-outputs-in-the-api/, [Accessed 02-09-2024]
[9] What is AWS CloudFormation? - AWS CloudFormation — docs.aws.amazon.com. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html, [Accessed 02-09-2024]
[10] Cauli, C.: Pre-deployment description logic-based reasoning for cloud infrastructure security (2022), University of Gothenburg. IT Faculty. Dissertation
[11] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code (2021), https://arxiv.org/abs/2107.03374
[12] Chiari, M., De Pascalis, M., Pradella, M.: Static analysis of infrastructure as code: a survey. In: 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). pp. 218–225 (2022). https://doi.org/10.1109/ICSA-C54293.2022.00049
[13] Guerriero, M., Garriga, M., Tamburri, D.A., Palomba, F.: Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 580–589 (2019). https://doi.org/10.1109/ICSME.2019.00092
[14] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.: Swe-bench: Can language models resolve real-world github issues? arXiv preprint arXiv:2310.06770 (2023)
[15] Lepiller, J., Piskac, R., Schäf, M., Santolucito, M.: Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 105–123. Springer International Publishing, Cham (2021)
[16] Opdebeeck, R., Zerouali, A., De Roover, C.: Control and data flow in security smell detection for infrastructure as code: Is it worth the effort? In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR). pp. 534–545 (2023). https://doi.org/10.1109/MSR59073.2023.00079
[17] Park, K., Wang, J., Berg-Kirkpatrick, T., Polikarpova, N., D'Antoni, L.: Grammar-aligned decoding (2024), https://arxiv.org/abs/2405.21047
[18] Qiao, F., Mohammadi, A., Cito, J., Santolucito, M.: Statically inferring usage bounds for infrastructure as code. arXiv preprint arXiv:2402.15632 (2024)
[19] Srivatsa, K.G., Mukhopadhyay, S., Katrapati, G., Shrivastava, M.: A survey of using large language models for generating infrastructure as code. arXiv preprint arXiv:2404.00227 (2024), https://arxiv.org/abs/2404.00227
[20] Tsai, Y., Liu, M., Ren, H.: Rtlfixer: Automatically fixing rtl syntax errors with large language models. arXiv preprint arXiv:2311.16543 (2023), https://arxiv.org/abs/2311.16543
[21] Ugare, S., Suresh, T., Kang, H., Misailovic, S., Singh, G.: SynCode: LLM generation with grammar augmentation (2024), https://arxiv.org/abs/2403.01632

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009