A Survey of using Large Language Models for Generating Infrastructure as Code

Kalahasti Ganesh Srivatsa[*1], **Sabyasachi Mukhopadhyay**[*1,2], **Ganesh Katrapati**[1], **Manish Shrivastava**[1]
1. Language Technologies Research Center, KCIS, IIIT Hyderabad, India.
2. Tejas Networks Ltd., Bangalore, India .
{kalahasti.ganesh, sabyasachi.m, ganesh.katrapati}@research.iiit.ac.in
sabyasachim@tejasnetworks.com
m.shrivastava@iiit.ac.in

Abstract

Infrastructure as Code (IaC) is a revolutionary approach which has gained significant prominence in the Industry. IaC manages and provisions IT infrastructure using machine-readable code by enabling automation, consistency across the environments, reproducibility, version control, error reduction and enhancement in scalability. However, IaC orchestration is often a painstaking effort which requires specialised skills as well as a lot of manual effort. Automation of IaC is a necessity in the present conditions of the Industry and in this survey, we study the feasibility of applying Large Language Models (LLM) to address this problem. LLMs are large neural network-based models which have demonstrated significant language processing abilities and shown to be capable of following a range of instructions within a broad scope. Recently, they have also been adapted for code understanding and generation tasks successfully, which makes them a promising choice for the automatic generation of IaC configurations. In this survey, we delve into the details of IaC, usage of IaC in different platforms, their challenges, LLMs in terms of code-generation aspects and the importance of LLMs in IaC along with our own experiments. Finally, we conclude by presenting the challenges in this area and highlighting the scope for future research.

1 Introduction

Infrastructure as Code (IaC) (Morris, 2016) has gained significance in modern software development as a mechanism for defining and managing IT infrastructure using code-based representations. With features like automation, scalability, consistency and version control (Francis, 2018), IaC is revolutionising the provisioning of infrastructure. Developers who are unfamiliar with this technology may find it challenging to create effective IaC templates. Moreover, the manual creation of infrastructure code can be time-consuming, error-prone and challenging to maintain, especially in complex environments.

Large Language Models (LLMs) have emerged as a new paradigm in NLP. Having been trained on a large amount of text for predicting the next word, with the given previous words and sentences by using an in-context learning mechanism, they have shown remarkable performance in downstream NLP tasks such as dialogue modelling, machine translation, question answering, text generation, sentiment analysis and so on.

Additionally, LLMs have demonstrated an ability in tasks related to code generation and validation. For instance, CodeParrot, CodeGen (Nijkamp et al., 2022), Llama (Touvron et al., 2023a), Google PaLm and OpenAI's GPT-3.5 Ouyang et al. (2022), GPT-4 are some of the models which are being studied and utilised for their code-generation capabilities. This leads to the promising possibility that IaC configurations could be generated automatically using LLMs, thereby addressing the problem of a steep learning curve, as well as, enabling users to understand the complexities and adjust parameters accordingly.

2 Background

In this section, we give a broad overview of IaC, with an introduction to well-known IaC platforms. This is followed by a brief overview of LLMs, focusing on the applications of LLMs for code generation and then a summary of relevant work regarding the incorporation of LLMs for IaC generation.

2.1 Infrastructure as Code (IaC)

Infrastructure as Code (IaC) is a software engineering and DevOps (Bass et al., 2015) practice that entails managing and provisioning infrastructure through code and automation (Punjabi and Bajaj,

* Authors contributed equally

2016) rather than manual operations. This defines and manages infrastructure elements both in physical and virtual machines, networks, storage and other resources using code scripts or configuration files. IaC tools are categorized (Duvall, 2011) as below two:

- **Provisioning:** Tools in this category provide infrastructure components for one or more cloud providers. Examples include HashiCorp's Terraform (Howard, 2022) and Pulumi[1].

- **Configuration management:** Tools in this category are used for installing and managing software on pre-existing infrastructure. Examples include Ansible[2], Chef[3] and Puppet[4].

Pre-IaC and need for IaC: Before IaC, IT had to rely on manual configuration and scripting, which was a tedious process and prone to errors. Additionally, there was a lack of consistency, making it difficult to maintain and troubleshoot. IaC has revolutionized IT management, making it more efficient and reliable than ever before.

A phased approach to the Evolution of IaC:

Early IaC methods relied on configurable scripting languages like Shell scripts (Greenberg et al., 2021) but had limitations in version control, modularity, and ease of use. To address these shortcomings, declarative configuration management tools were introduced which also had their limitations in terms of scalability, flexibility, and cloud support. With the advent of cloud computing, the need for dynamic and scalable infrastructure provisioning became apparent. This led to the emergence of infrastructure orchestration tools such as Terraform, Ansible, and Pulumi, enabling on-demand provisioning in cloud environments.

IaC, which employs a descriptive model, follows three key steps: (i) Developers use Domain-Specific Language (DSL) (Shambaugh et al., 2016) to define the configuration state in a file. (ii) This configuration file is then transferred to a server, code repository, or API. (iii) The system configures the infrastructure based on the instructions in the transferred file, ensuring reliable versioning and deployment.

[1] https://github.com/pulumi/pulumi
[2] https://www.ansible.com
[3] https://www.chef.io/products/chef-infra
[4] https://www.puppet.com/

Approaches of IaC: **Declarative (functional)** approach outlines the intended state, allowing the system to perform necessary steps without specific syntax. Examples include Terraform, AWS CloudFormation. **Imperative (procedural)** approach provides specific commands in the correct sequence for desired results. Example includes Ansible, Pulumi.

Benefits: IaC offers numerous industry benefits (Humble and Farley, 2010; Cois et al., 2014), including automation, consistency, rapid deployment, transparency, version control, scalability, reusability, modularity and immutability. These benefits help in reducing errors, configuration drifts, efficient deployment, tracking the changes, rollbacks, traffic management and reusable code across various environments by ensuring stability without modifying the deployed instances.

Challenges: Though IaC offers the above benefits, it comes with its own set of challenges (Siebra et al., 2018) such as an increase in the complexity due to intricate code for complex configurations, consistency, compatibility, and dependencies across various platforms. It also involves expertise in handling scalability, proper version tracking, tool selection and security aspects.

Some of the popular IaC tools are ***(i) Terraform*** an open source project, that uses HashiCorp Language (HCL), ***(ii) Ansible***, open source project automates provisioning using YAML, ***(iii) Pulumi***, a tool with the flexibility for using general programming languages like C, C++, Python etc., ***(iv) Kubernetes*** etc.

2.2 Large Language Models (LLM)

Language modelling (LM) advances machine intelligence by modelling the generative likelihood of word sequences and predicting future token probabilities. LMs have garnered substantial attention and transitioned through four distinct developmental stages starting from Statistical LM, progressing through Neural LM namely RNN (Bengio et al., 2000; Mikolov et al., 2010, 2011), LSTM (Hochreiter and Schmidhuber, 1997) and Transformers (Vaswani et al., 2017), further advancing into Pretrained LMs like BERT (Devlin et al., 2018) and culminating in the emergence of LLMs. The Transformer architecture, based on a self-attention mechanism, allows for efficient parallelization and handling of long-range dependencies is a major breakthrough for LLMs. LLMs such as GPT (Radford et al., 2018) and RoBERTa (Liu et al., 2019) mod-

els exhibit high potential in performing NLP downstream tasks.

2.3 LLMs for Code Generation Task

Code generation, synthesis and summarization tasks have been promising research in recent times with the increased capabilities of LLMs. As per the survey of Xu et al. (2022), there are three ways to pre-train a code generation model.

Decoder-Based LM: An auto-regressive, left-to-right model also called Causal Language Model (CLM) performs well on code generation and completion tasks. In this, the model predicts the next token based on the previous token. Codex (Chen et al., 2021), a GPT-3 (Brown et al., 2020) based 12 billion parameters model which has been pre-trained on 159 GB of code samples from 54 million GitHub repositories, solved 28% of HumanEval. According to a study by Chen et al., scores have improved with the use of repeated sampling or pass, a concept in which the model is given 100 chances and if it can generate 1 correct sample out of 100 samples, the model has solved the task. CodeParrot[5](Tunstall et al., 2022) trained on 25-30B tokens of Google BigQuery data, evaluated on HumanEval where CodeParrot 110M(small) outperformed the CodeParrot 2B(large). CodeGen (Nijkamp et al., 2022), proposed Multi-Turn program synthesis model trained on The Pile (Gao et al., 2020), BigQuery[6] which has natural language, code, configuration files in its dataset and BigPython datasets. CodeGen-Multi, fine-tuned on Python files and termed as Mono model improved the program synthesis task substantially. Their study says that as there is an increase in the size of the model, there is an increase in the overall performance also. A few more examples of decoder-based models are LLaMA & LLaMA-2 proposed by Touvron et al. (2023a,b) are trained on public GitHub data available on Google BigQuery to generate a code based on natural language description. This is evaluated on HumanEval and MBPP (Austin et al., 2021) datasets. LLaMA-2 outperformed LLaMA1 and other general models. As per their study further fine-tuning on code-related data would increase the capability of the model. GitHub Copilot is an AI tool developed by GitHub along with OpenAI that takes natural language input and generates, completes and comments the code. Instruct GPT (Ouyang et al., 2022) uses Reinforcement Learning with Human Feedback (Christiano et al., 2017; Stiennon et al., 2020) along with models like code-davinci-002, text-davinci-003 that has shown their program synthesis abilities. PaLM (Chowdhery et al., 2022) model takes natural language (NL) prompts and assists in code generation and completion tasks.

Encoder-Based LM: Auto-Encoding model performs well on code detection and classification tasks by utilising information bi-directionally. CodeBERT (Feng et al., 2020) which is a bimodal pre-trained model trained on natural language (NL) and programming language (PL), achieved state-of-the-art (SOTA) performance on NL-PL downstream tasks by outperforming RoBERTa in a zero-shot setting. CuBERT (Kanade et al., 2020) is another model that outperformed other models with lesser data and fewer epochs.

Encoder-Decoder Based LM: CodeT5 (Wang et al., 2021) which extends T5 (Text-To-Text-Transfer-Transformer) (Raffel et al., 2020) works on the objectives of masked span prediction, denoising sequence reconstruction, and masked identifier prediction with a bimodal dual generation, encourages a better alignment between NL and PL. This outperformed the previous SOTA model PLBART (Ahmad et al., 2021) on all generation tasks including code summarization, text-to-code generation, code-to-code translation, and code refinement.

3 IaC using LLM: Related Works

There are some related works where IaC has been generated by LLMs.

3.1 Ansible-YAML Generation by LLMs

1. **Ansible-YAML file generation by open-source models:** The study in Pujar et al. (2023) explores the use of LLMs transformer-based models to generate Ansible-YAML code from Natural Language prompts, providing an AI assistant for users to increase productivity. IT infrastructure relies on YAML files for defining and configuring crucial elements. It begins by learning from a sizable amount of YAML and Ansible-YAML data, curated from multiple data sources, including GitHub, Google BigQuery, GitLab and Ansible Galaxy (Heap and Heap, 2016) and deduplicated using the exact match method. The

[5] https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot

[6] https://cloud.google.com/bigquery

curated dataset contains closely 1.1M Ansible tasks, YAML playbooks, and nearly 2.2M other generic YAML files. Their pre-trained models WISDOM-ANSIBLE and WISDOM-YAML are trained on CodeGen architecture's checkpoints that contain Ansible-YAML and YAML files to improve the understanding of the syntax and semantics of YAML files. The ANSIBLE-Galaxy dataset is a collection of high-quality files developed and approved by the Ansible community and is utilized to fine-tune the pre-trained models for Ansible-YAML generation tasks. The major contributions of their work are: (i) Providing a formal definition of the problem when applying code generation to Ansible-YAML, (ii) Creating YAML and Ansible-YAML datasets for pre-training and fine-tuning code generation tasks, (iii) Reformalizing the problem of generating Ansible YAML into a code completion task with novel prompts (iv) Proposed two novel metrics designed specifically for Ansible-YAML. WISDOM-ANSIBLE and WISDOM-YAML model's training code is based on Huggingface Transformer library (Wolf et al., 2020), with the provided model checkpoints and tokenizers on the YAML data for 9 epochs on 16 A100 GPUs with 80GB of memory, batch size of 32, $5*10^{-5}$ learning rate and context window of 1024 along with bf16 data type to fasten the training process. The task utilizes a task description consisting of Natural Language prompt X and Ansible-YAML context script C that can generate two kinds of output either a full playbook or a task in the playbook Y. They defined a probabilistic distribution of the Ansible snippet Y given X and C as $p(Y|X, C)$ and the best possible Ansible task snippet is denoted by $\hat{y} = argmaxp(Y|X, C)$

Thus, the generated Ansible YAML files are evaluated using these 4 metrics ***Exact Match***, ***BLEU*** (Papineni et al., 2002) and ***Ansible Aware*** which uses the Ansible YAML syntax knowledge to compare the modules as well as ***Schema Correct*** which measures the correctness of the result. Results indicate that pre-trained models WISDOM-ANSIBLE and WISDOM-YAML outperform CODEGEN and CODEX(Codex-Davinci-002) on all 4 metrics. Also, their fine-tuned models show an increase in performance with respect to the pre-trained models.

2. **Ansible-YAML File Generation by Language Models(LM):** A similar study has been done by Kawaguchi et al. (2022) where the designed system uses an LM that has been tuned on Ansible playbooks semantics to suggest potential commands based on the status of the current input (i.e., code completion function). The main goal of this work is to prevent network downtime caused by misconfigurations by using an automation tool. This paper proposes an architecture with a client and server paradigm. The client program has an editor for creating Ansible playbooks and a code completion function using LM. In their study, they state that the previous work on completion using LM can only output the candidates of the succeeding command based on the current input command. Suppose when the LM is trained with both YAML configurations, "tasks:*, name:*, yum:*" and "tasks:*, name:*, template*", with * indicating variable string, in the previous models when the model is given input as "tasks:*", it outputs only its immediate command "name:*" which is a unigram prediction. But in the proposed system, if the model is given an input of "tasks:*" it outputs the complete Ansible command "name:*", "name:*, yum:*", "name:*, template:*" by recursively using the output candidates of the language model as the input to generate the next output. This candidate list is ordered by the appearance frequency in training data.

   Upon YAML configuration file completion, operator easily sends it to the server application which utilizes Ansible to compile and activate the settings on the target network equipment, while the client program supporting the operator's configuration. The evaluation demonstrates improved accuracy as the number of input commands grows.

3.2 LLMs used in DevsecOps

1. **Static Code Analysis of IaC:** The primary goal of Petrović (2023a) is to leverage ChatGPT for static code analysis in order to target different IaC standards, with a particular emphasis on Terraform and Ansible in the context of DevSecOps. As a first step, the user

must choose and upload the desired archive that contains the IaC scripts in charge of deploying the underlying infrastructure. Each individual IaC file is read and converted as a string to form the pre-defined question for the ChatGPT in the form of the question "Find security flaws in $filetype$ script:$contents$". Here, the first parameter serves as a placeholder for the IaC-related file type and the second one contains the script's actual content. Following the construction of the inquiry, a ChatGPT request is submitted via Python API and then a summary of the received responses for each of the IaC files is included in an HTML table that serves as the final output of the solution. Each entry represents a different file from the archive, along with ChatGPT's summary with probable defects and suggestions for resolving it.

2. **Run-time Analysis of Server Logs:** The study Petrović (2023b) primarily focuses on leveraging a machine learning approach using server log analysis to detect suspicious activities in runtime security through traffic-related data. Using ChatGPT, this novel technique utilizes context i.e labelled data and queries, in conjunction with log entries to assess the suspicious activities. An essential aspect is the volume of data required for training new prediction models when log structures change. This study explores the innovative application of ChatGPT and LLM for log analysis, aiming to reduce the need for extensive training data by adapting a pre-trained model to yield satisfactory results. The user input comprises question-context pairs, where the context consists of group of labelled log entries, serving as an input to ChatGPT for pattern extraction. With this input, ChatGPT labels network traffic records and explains the meaning of log records, including its underlying protocol, data exchange, communication flow and network traffic details. Finally, the user receives notifications about any detected suspicious traffic or activities.

3.3 IaC Generation through ChatGPT Queries

1. **ChatGPT for DevOps:** The blog Harvey (2023) demonstrates use of ChatGPT for DevOps by prompting the tool to show its ability to generate accurate Python and Bash scripts. With the results generated, the author queried the task-specific Terraform configuration using Chatgpt, finding that OpenAI Playground's configuration was identical to manual configurations and better than ChatGPT's results. However, in queries about broader DevOps concepts, results were less accurate, resembling Google search results. Nevertheless, utilizing ChatGPT or OpenAI Playground proved helpful in understanding and getting a way out for some of the CI/CD and code debugging tasks.

2. **SSO in Kubernates Configuration Generation:** In the blog Entzmann (2023), ChatGPT was utilized to generate Kubernetes configurations for Azure, specifically related to Single Sign-On (SSO) configurations using Azure Active Directory. ChatGPT successfully generated a sample kube-API server YAML file and detailed its parameters related to SSO and also a sample kube-config file for use with the kube-apiserver. Although it provided a valid solution using the oidc-login plugin of kubectl, an error surfaced when connecting the oidc plugin with HTTPS to Azure, suggesting an extra, non-existent parameter "-https." When challenged, ChatGPT acknowledged the mistake and advised using a reverse proxy, that is different from the desired configuration parameters.

3.4 IaC Generation Tools with LLM

1. **Infracopilot** (Masolo, 2023) is a cutting-edge IaC editor that uses LLM to understand user intent and Klotho [7], an open-source engine, to revolutionize cloud infrastructure development and management by providing unparalleled intelligence, flexibility, agility, consistency, clarity and reliability. It includes components like API/Orchestrator, Intent Parser, Visualization Engine, and Discord Bot to communicate requests to the service. LLMs extract user intent, sent to intent corrector, that confirms, corrects, convert it into JSON, and update intents. Klotho Engine generates and verifies architecture, including low-level components like VPCs, subnets, security groups, and IAM policies.

[7] https://klo.dev/announcing-infracopilot/

2. **K8sGPT** (Singh, 2023) is a tool that scans and diagnoses Kubernetes clusters using SRE-encoded analyzers like PodAnalyzer, pvcAnalyzer, rsAnalyzer, serviceAnalyzer, eventAnalyzer and ingressAnalyzer providing efficient troubleshooting. It features default and customizable analyzers like hpaAnalyzer and pdbAnalyzer which are activated for specific needs. Filters control resource analysis, enhance analysis capabilities with commands like "k8sgpt filters list" displaying available filters and "k8sgpt filters add/remove" for adding or removing multiple filters. "k8sgpt integration activate/deactivate" activates or deactivates the integrations with tools like Trivy.

3. **Pulumi AI**[8] introduced an AI Assistant to accelerate cloud infrastructure development by leveraging LLMs and GPT, aiding intelligent resource identification and interaction within Pulumi cloud. Pulumi Insights (insights, 2023) deploys generative AI and LLMs for infrastructure enterprise, analytics and offering key elements to manage cloud footprints. The Pulumi resource supergraph provides metadata and links across cloud infrastructure, assisting in cloud architecture design, allowing users to visualise data for cost, compliance, and operations using preferred BI tools.

4 IaC Code Generation Process

This section outlines a template for the generation of Infrastructure as code through LLMs using Terraform as an example and some basic results.

4.1 Pre-Training

Pre-trained models (Qiu et al., 2020) serve as foundational components for diverse downstream tasks, trained on large benchmark data to facilitate easy fine-tuning in various applications and tasks, enabling comprehensive knowledge capture and semantic representation for tasks like text generation etc.

4.2 In-Context Learning

In-context learning, also often called "zero-shot" or "few-shot" learning, utilizes the pre-training data to provide context and task examples, enabling models to infer expected behaviour and produce appropriate responses. This concept helps in rapid experimentation without fine-tuning model settings in unlabelled data situations.

For generating ***"text to terraform configuration"*** with a few-shot setting in a model, it utilizes the text prompt for specific configuration and context files with sample terraform configuration as an input and the model keeps repeating until all blocks of terraform are generated. In our example, we used a model with a few-shot setting in LLM like GPT 3.5-Turbo for generating "text to terraform configuration", and used context files with sample terraform, provider and resource blocks along with a text prompt given for a specific configuration and the model generates the configuration for the block, repeating until all blocks are generated.

This offers numerous advantages, particularly in situations that lack labelled data or require UI/API interaction, allowing rapid experimentation without fine-tuning model settings.

4.3 Data Collection and Fine-Tuning

- **Instruction Fine-Tuning:** It is imperative for LLMs when generating IaCs with an IaC prompt due to its misalignment with the pre-trained language modelling objective of predicting the next token, as it might exhibit unanticipated behaviours like generating harmful or fabricating content. Hence using fine-tuning approach is essential for these tasks that involves defining the task, architecture, loss function, and data selection. Thus our work, generating "Infrastructure as Code" configuration files from English text empowers engineers to refine configurations post-review.

  To initiate fine-tuning, data pertinent to the task must be collected. For Terraform IaC, Google BigQuery's GitHub dataset with .tf extensions serves as a substantial source, comprising 164MB of Terraform data across 23839 files from a 1.5TB corpus. After eliminating duplicates, we pre-processed, and split the data into 75% training and 25% validation sets. In our Code-parrot's experiment we utilized 19071 files for training and 4768 for validation.

  The pre-trained model, enriched with linguistic knowledge from vast code and text data, is then adapted for fine-tuning. The adapted model involves continuous training on task-specific data that contain terraform prompts and configuration examples in our work. We

[8] https://www.pulumi.com/ai/

adjust the model's pre-trained parameters to better suit the task, using gradient descent to update parameters based on task-specific data loss. Performance enhancements are achieved by adjusting hyper-parameters like training epochs, batch size, learning rate, and weight decay. The refined model is preserved for analysis. In the Code-parrot example, fine-tuning spanned 20000 epochs by utilizing the above datasets.

4.4 Evaluation

In order to benchmark generative models for code, samples are typically compared to a reference solution for Functional correctness (Chen et al., 2021) A similar approach is used for IaC evaluation too.

- **Functional correctness by exact match:** This metric compares the function of a generated configuration file to a reference solution by ensuring the same functionality across different setups. A Terraform uses LLM-generated configuration file to develop and create a JSON execution plan if the generated configuration file compiles and matches with the reference solution plan and considered as a success. This implies that even the slightest error in the configuration file generated is considered as a failure. The reference dataset contain tasks that have a natural language description and a desired configuration. Terraform 1.4.6 was used for this activity.

A task is is a JSON-formatted text file that describes one or more containers that forms our application.

The evaluation dataset covers all features of Terraform configuration files including jobs from GCP, AWS, Azure and virtual machines with task complexity varying to maximize LLMs capabilities. To compare two plans, JSON file details need anonymization, and multiple samples are generated per task to determine the average success rate due to the stochastic nature of code/config generation.

4.5 Experiments and Results Analysis

As a part of the survey, we performed experiments with 4 GPUs of Nvidia GeForce RTX 2080 Ti (11GB) and reported our scores on the models CodeParrot small (110M) and GPT-3.5-turbo models with a single sample and multiple sample configuration generation for 49 different tasks for AWS service providers. These tasks are a collection of various configuration areas on the AWS cloud through Terraform. The Table 1 summarizes the results obtained by configuration generation and evaluation by functional correctness by exact match with human generated terraform configuration for two models.

	Single sample	**50 samples**
GPT-3.5 turbo	59.18%	56.81%
CodeParrot (Small 110M)	8.2%	8%

Table 1: Experimental Results

1. **GPT3.5-turbo:** Model used in-context learning to generate 1-sample and 50-sample configurations for all 49 tasks in AWS provider at temperature setting 0.2. The generated configurations are evaluated against human generated configurations using functional correctness by exact match. It obtained an average success rate(accuracy) of 59.16% with 1-sample and 56.81% with 50-samples, by calculating the mean of success rate of total number of samples generated under each task and the final score is the average of total success rate of all 49 tasks in AWS provider.

2. **Codeparrot:** In this also, model generated 1-sample and 50-sample configurations for all 49 tasks in AWS provider and evaluated the generated configurations with human generated configurations using functional correctness by exact match. The same methodology mentioned above in GPT3.5-turbo for calculating the average success rate(accuracy) is followed and obtained an accuracy of 8.2% with 1-sample and 8% accuracy with 50-samples.

In summary, GPT-3.5-Turbo outperforms the CodeParrot model due to its extensive and diverse training dataset, as well as its inherent adaptability through fine-tuning that collectively contributed to GPT-3.5 Turbo's remarkable performance.

5 IaC using LLM: Safety and Ethical Considerations

This section outlines the safety and ethical considerations of LLM-generated configurations in production environments while proposing potential resolutions.

5.1 Safety Concerns:

- **Security Risks:** Unvalidated IaC can lead to vulnerabilities like misconfigured databases, lax security, and exposed secrets.
- **Over-Reliance:** Blindly relying on LLM generated configurations is risky, understanding them is crucial.
- **Resource Overutilization:** Poorly tuned IaC can create unnecessary resources, that results in cost overruns and also comes with the risk of over-provisioning leading to environmental expenses.
- **Updates and Maintenance:** LM's struggle with real-time changes to cloud platform etc., leading to outdated or ineffective setups.

5.2 Best Practices:

- **Review and Validate:** Continuous evaluation of IaC for performance, security, and compliance through manual and automated reviews.
- **Test in Isolated Environments:** Predeployment testing in sandbox environments helps uncover issues overlooked during code review.
- **Version Control:** Store IaC in version-controlled repositories for collaboration, auditing, and easy change reversals.
- **Educate the Team:** Ensure your team understands IaC principles and leverages LLMs as tools, to enhance expertise by staying updated with the platform and technology.
- **Limit Permissions:** Safeguard production deployments by restricting LLM access and having human supervision in your CI/CD or automation framework.
- **Feedback Loops:** Create feedback mechanisms to refine LLM training and prompts based on IaC deployment results.

5.3 Ethical Considerations:

- **Transparency:** LLMs that conceal their decision-making process can cause due diligence concerns when using IaC models, as stakeholders often seek more details to understand the decisions.
- **Accountability:** Clear responsibility lines are essential for preventing large-scale failures or breaches in LLM-generated IaC, as determining accountability for defective and unsecure systems is complex.
- **Bias and Fairness:** LLMs trained on suboptimal data may perpetuate errors and overlook organizational or cultural nuances, creating IaC suitable for one context but not another.
- **Dependency and Vendor Lock-In:** Excessive reliance on a single LLM for IaC risks vendor lock-in, reduced flexibility, and potential cost escalation.

5.4 Recommendations:

- **Human-in-the-loop:** Incorporating human judgment for critical infrastructure decisions.
- **Data Diversity:** Ensuring varied training data for comprehensive best practices.
- **Regular Audits:** Periodically reviewing LLM-generated IaC for bias and inefficiencies.
- **Stakeholder Education:** Ensuring stakeholders understand LLM capabilities and limitations to manage expectations effectively.

Based on our experimental study and detailed analysis from various references, we conclude that LLMs have the ability to generate IaC with great efficiency, but they must be employed carefully with awareness of their ethical ramifications.

6 Challenges and Future Study

Limited GitHub training data and Terraform representation may produce syntactically correct but erroneous code. LLMs lack awareness of current practices and security, risking data exposure. They may offer unsuitable solutions, misaligned with use cases. API updates affect code quality. Testing complex LLM-generated IaC complicates deployment. Lack of best practices and comments hampers modifications. Integration issues with DevOps tools can lead to cost inefficiencies.

To mitigate challenges, we can implement a comprehensive review process by engaging domain experts, and rigorously testing generated infrastructure code. In future, we can also use LLMs as assistants in multi-turn IaC Chatbots with automatic

validations. Also we plan to expand our experiments using 1000 samples per task, comparing our results with open and closed-source models.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A software architect's perspective*. Addison-Wesley Professional.

Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in neural information processing systems*, 13.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30.

Constantine Aaron Cois, Joseph Yankel, and Anne Connell. 2014. Modern devops: Optimizing software development through effective system interactions. In *2014 IEEE international professional communication conference (IPCC)*, pages 1–7. IEEE.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Paul M Duvall. 2011. Continuous delivery: Patterns and antipatterns in the software life cycle. *DZone refcard*, 145.

Benoît Entzmann. 2023. Chatgpt vs devops. Accessed: 2023-02-01.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Erik Francis. 2018. Infrastructure as code: Everything you need to know. Accessed: 2023-08-10.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. 2021. Unix shell programming: the next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 104–111.

Caoimhe Harvey. 2023. Using chatgpt for devops. Accessed: 2023-02-09.

Michael Heap and Michael Heap. 2016. Advanced ansible. *Ansible: From Beginner to Pro*, pages 137–157.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Michael Howard. 2022. Terraform–automating infrastructure as a service. *arXiv preprint arXiv:2205.10676*.

Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

insights. 2023. Pulumi insights: Intelligence for cloud infrastructure. Accessed: 2023-04-13.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR.

Mamoru Kawaguchi, Kimihiro Mizutani, and Nobukazu Iguchi. 2022. An implementation of misconfiguration prevention system using language model for a network automation tool. *IEICE Proceedings Series*, 72(S5-8).

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Claudio Masolo. 2023. Infracopilot, a conversational infrastructure-as-code editor. Accessed: 2023-05-31.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Interspeech*, volume 2, pages 1045–1048. Makuhari.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2011. Extensions of recurrent neural network language model. In *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5528–5531. IEEE.

Kief Morris. 2016. *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.".

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Nenad Petrović. 2023a. Chatgpt-based design-time devsecops. *preprint*.

Nenad Petrović. 2023b. Machine learning-based runtime devsecops: Chatgpt against traditional approach. *preprint*, pages 1–5.

Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matt Jones, Alessandro Morari, et al. 2023. Automated code generation for information technology tasks in yaml through large language models. *arXiv preprint arXiv:2305.02783*.

Rahul Punjabi and Ruhi Bajaj. 2016. User stories to user reality: A devops approach for the cloud. In *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 658–662. IEEE.

Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 416–430.

Clauirton Siebra, Rosberg Lacerda, J Peixoto, I Cerqueira, F Da Silva, and A Medeiros. 2018. From theory to practice: The challenges of a devops infrastructure as code implementation. In *ICSOFT 13th 2018, International Conference on Software Technologies. Porto: Portugal July*, pages 26–28.

Jasbir Singh. 2023. Unlocking the power of kubernetes with k8sgpt. Accessed: 2023-06-18.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. 2022. *Natural Language Processing with Transformers, Revised Edition*. O'Reilly Media, Incorporated.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.