Leveraging Large Language Models for Generating Infrastructure as Code: Open and Closed Source Models and Approaches

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering
by Research

by

Kalahasti Ganesh Srivatsa
2021701036
kalahasti.ganesh@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
June 2024

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Leveraging Large Language Models for Generating Infrastructure as Code: Open and Closed Source Models and Approaches" by Kalahasti Ganesh Srivatsa, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Advisor: Dr. Manish Shrivastava

This work is dedicated to my family, friends and guide for their love, boundless support and encouragement.

Acknowledgments

I hereby express my sincere gratitude to my supervisor, Dr. Manish Shrivastava, for his unwavering support throughout the course of my Master's thesis. I am immensely grateful for his guidance and motivation, starting from the first day of our discussion, which helped me formulate the problem statement and gain a deeper understanding of the research area. His technical expertise and valuable suggestions helped shape this thesis at every stage. I also gained his personal and ethical insights into research, which will be invaluable for my future endeavors.

I would like to thank my collaborator, Sabyasachi Mukhopadhyay, for this thesis. His help and efforts has been so valuable in completing this thesis. I would also like to acknowledge my colleagues and senior researchers in the lab who constantly supported me during my entire Master's program. Our countless hours of intense discussions provided me with a clear understanding of best research practices.

I express my heartfelt thanks to IIIT Hyderabad for providing the coursework that laid the foundation for my research work. I thank all my course instructors, faculty and teaching assistant staff for their insightful lectures, assignments, and quizzes. The principles I grasped in my courses not only bolstered my technical abilities but also laid the groundwork for my research endeavors, building upon the core projects I successfully executed. I am also grateful to my friends Srivathsa, Sarvani, Ashok, Lokesh, Gopichand, Pavan for their great support in this journey.

Finally, I express my deepest gratitude to each and every of my family members and friends for their constant support. The immense support I received from my family and wife was instrumental in completing my Master's degree.

Abstract

Infrastructure as Code (IaC) has emerged as a groundbreaking paradigm in the Software Engineering community and tech industry, offering unparalleled efficiency and scalability. By leveraging machine-readable code, IaC streamlines the management and provisioning of IT infrastructure, showing a way in a new era of automation, consistency, reproducibility, and error reduction across diverse environments. Despite its transformative potential, orchestrating IaC remains a labourious work, demanding a specialized skills and significant reduction in the manual efforts.

Recognizing the importance for automation in today's fast-paced industry landscape, this survey investigates the feasibility of harnessing Large Language Models (LLMs) to address the challenges in IaC orchestration. LLMs, a large neural networks based models, exhibit remarkable language processing capabilities and have demonstrated their efficiency in executing various instructions across a wide range of applications. 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.

The major focus of this thesis delves into the intricacies of IaC, exploring its utilization across different platforms and explaining the associated challenges. It examines the ability of LLMs in code-generation task, highlighting their potential significance in automating IaC workflows. Based on our experimental findings, we illustrate the importance of LLMs in the context of IaC and share insights of the possible applications in this domain.

In conclusion, we highlight the challenges inherent in leveraging LLMs for IaC automation and emphasise the vast potential for future research endeavors to address these challenges that helps further in enhancing the efficacy of IaC orchestration and also to build an end-to-end system that comprehends the users intent and motivation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Language serves as a defining characteristic of human communication, enabling the nuanced expression of thoughts and ideas. Unlike humans, machines lack the inherent capacity to comprehend and communicate through language without the aid of Artificial Intelligence (AI) tools and techniques. Bridging this gap between human linguistic capabilities and machine communication has posed a persistent challenge for researchers striving to endow machines with human-like language understanding and expression.

Amidst the ever-growing complexity and scale of modern infrastructure systems, the demand for efficient and scalable approaches to infrastructure provisioning and management has reached unmatched levels. In recent years, the advent of large language models (LLMs), empowered by advanced AI technologies such as OpenAI's GPT-3.5 and GPT-4, has emerged as a transformative force in generating human-like text and code across diverse domains.

Infrastructure as Code (IaC) has emerged as a foundation of modern software development, offering a mechanism for defining and managing IT infrastructure through code-based representations. With its inherent features including automation, scalability, consistency, and version control, IaC is reshaping the landscape of infrastructure provisioning. Yet, developers unfamiliar with this technology may encounter difficulties in crafting effective IaC templates. Moreover, the manual creation of infrastructure code is prone to errors and can be unmanageable to maintain, particularly in intricate environments.

The rise of LLMs signifies a paradigm shift in Natural Language Processing (NLP), demonstrating exceptional performance across various downstream tasks such as dialogue modeling, machine translation, question answering, and sentiment analysis. Additionally, LLMs have showcased their prowess in code generation and validation tasks, exemplified by models such as CodeParrot, CodeGen, Llama, Google PaLm, GPT-3.5, and GPT-4. This convergence of capabilities presents an intriguing possibility wherein LLMs could automate the generation of IaC configurations, mitigating the steep learning curve associated with manual creation and empowering users to navigate complexities and adjust parameters effectively.

The advantages of harnessing automation tools like LLMs for IaC generation are manifold. LLMs offer precise control over infrastructure elements, facilitating parameter adjustments for complex setups and scenarios. Furthermore, they enhance transparency in code understanding, providing clear and

unambiguous information about infrastructure. IaC generated by LLMs is optimized for performance and other considerations, offering flexibility in infrastructure design and integration with custom applications. Importantly, leveraging LLMs for IaC generation minimizes vendor lock-in concerns and reduces the learning curve for engineers, thus aiding a deeper understanding of infrastructure management and cloud platforms.

In light of these considerations, this thesis aims to explore the potential of LLMs in automating IaC generation, examining their capabilities, advantages and implications for modern infrastructure provisioning and management. Through theoretical analysis and empirical investigation, this study seeks to shed light on the efficacy of LLM-driven automation in enhancing infrastructure agility, efficiency, and scalability, while also identifying the scope for further research and innovation in this rapidly growing field.

1.1 Motivation

The exponential growth of digital infrastructure has driven the need for efficient and scalable management solutions. However, the manual creation of Infrastructure as Code (IaC) templates poses challenges in terms of complexity, error-proneness, and steep learning curves for developers. Additionally, while LLMs have demonstrated prowess in various natural language processing tasks, their potential in automating IaC generation remains largely untapped. Harnessing LLMs for this purpose holds the promise of streamlining infrastructure provisioning processes, reducing human error, and enhancing scalability. By automating IaC generation, organizations can achieve greater agility, cost-effectiveness, and adaptability in managing their infrastructure. Therefore, exploring the integration of LLMs into IaC workflows presents an exciting opportunity to revolutionize infrastructure management practices and unlock new levels of efficiency and innovation in the digital era.

1.2 Infrastructure as Code (IaC)

Virtualized resources and cloud technologies simplify infrastructure replacement, offering automation and reduced provisioning time compared to previous systems. A new mindset is needed to adopt Cloud Age technologies, adopting rapid advancements to reduce risk and improve quality. A shift in strategy and fresh perspectives on change and risk is necessary for “Infrastructure as Code" system management.

Infrastructure as code (IaC) is a software engineering and DevOps [4] practice that entails and manages computer data centers using machine-readable definition files, utilizing declarative approaches instead of interactive configuration tools or physical hardware setup. This method manages both physical and virtual computers and configuration resources. IaC is an infrastructure automation approach based on software development techniques, focusing on consistent, reproducible processes for system vision and modification, testing, and application.

1.3 Large Language Models, its Importance and Role in IaC

Large Language Models (LLMs) represent a paradigm shift in the field of Natural Language Processing (NLP), harnessing the power of extensive text data for predictive learning. Trained on vast corpora of text, these models excel in predicting subsequent words and sentences within a given context, leveraging an in-context learning mechanism. Their remarkable performance across various downstream NLP tasks, including dialogue modeling, machine translation, question answering, text generation, and sentiment analysis, has accumulated significant attention.

Beyond traditional NLP applications, LLMs have demonstrated a remarkable aptitude for tasks related to code generation and validation. This includes CodeParrot, CodeGen, Llama, Google PaLm, as well as OpenAI's GPT-3.5 and GPT-4. These models leverage their language understanding capabilities to generate code snippets, validate syntax, and even offer insights into best programming practices.

This convergence of LLMs and code generation presents a promising avenue for automating the creation of Infrastructure as Code (IaC) configurations. By harnessing the predictive skill of LLMs, organizations can potentially mitigate the steep learning curve associated with IaC adoption. Additionally, LLM-generated IaC configurations offer users a means to comprehend the complexities of infrastructure provisioning and adjust parameters with greater ease and efficiency.

As a conclusion note, the integration of LLMs for automatic IaC configuration generation holds a tremendous potential, streamlining the deployment process, enhancing productivity, and fostering greater accessibility to infrastructure management practices. This innovative approach not only addresses the challenges present in the manual IaC development but also empowers organizations to adapt swiftly to evolving infrastructure requirements in today's dynamic technological landscape.

1.4 Thesis Contribution

The main objective of this thesis is to provide a thorough study of using LLM's for IaC generation by addressing the need and importance of this work in the industry along with its benefits and associated challenges. As a part of this research, we hope to provide a valuable insights of importance and scope of our this research in the DevOps space. This thesis also emphasises about the benifits, major challenges that we faced as a part of our research and also provides the detailed study of the safety and ethical considerations of IaC using LLLM's.

- We provide a detailed study in the area of IaC and usage of LLM's for various IaC tools.
- This thesis outlines the experimentation and evaluation details of using LLM's for generating Terraform configurations which is one of the popular IaC tools.
- Additionally, it also explores various generative language models and finetuning approaches which gained us a good volume of knowledge and understanding of our work.

1.5 Thesis Workflow

The thesis is structured as follows: Chapter 2 briefly discusses about the existing works which are very few in this area. Chapter 3 demonstrates the details of Infrastructure as Code (IaC), Evolution of IaC, role of IaC in the DevOps practice along with the benifits and challenges of IaC. Chapter 4 briefs about the Language Models, evolution of Large Language Models (LLM's), importance of Code Generation Models and adaption of those models for our work. Chapter 5 briefs about the dataset and its pre-processing details, training, fine-tuning approaches utilized and also the model evalution details, the experimental setup, results and its analysis. Chapter 6 provides a thorough study on safety and ethical considerations for IaC using LLM's. Chapter 7 details the conclusions and future works.

Chapter 2

Literature Review

Natural Language Processing (NLP) is a branch of Artificial Intelligence that helps machines understand and process human language so they can carry out activities automatically. Machine translation, summarization, question-answering systems, sentiment analysis, categorization, and spell checking are a few downstream uses of NLP. Text is normally entered into NLP systems. Initially, the text input is converted into a vector representation of actual values. Based on the downstream application, this vector is analysed by a model that generates either a class label or text.

2.1 IaC using LLM: Related Works

This chapter discusses about the some of the important related works where IaC has been generated by LLMs and its impact in the industry.

2.1.1 Ansible-YAML Generation by LLMs:

1. **Ansible-YAML File Generation by Open-Source Models:** The study in [43] 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 [20] and deduplicated using the exact match method. The curated dataset contains approximately 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:

- Providing a formal definition of the problem when applying code generation to Ansible-YAML.
- Creating YAML and Ansible-YAML datasets for pre-training and fine-tuning code generation tasks.
- Reformalizing the problem of generating Ansible YAML into a code completion task with novel prompts.
- Proposed two novel metrics designed specifically for Ansible-YAML.

WISDOM-ANSIBLE and WISDOM-YAML model's training code is based on Huggingface Transformer library [61], 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(NL) 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*** [40] 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 for 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 [28] where the designed system uses an LM that has been tuned on Ansible playbook's 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 through the use of an automation tool. The 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 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.

Once the YAML configuration file is complete, the operator can effortlessly push it to the server application. The server program uses Ansible to compile the received YAML file and activate the setting on the target network equipment, while the client program supports the operator's configuration. The evaluation shows that the system can propose more correct candidates as the number of input commands increases.

2.1.2 LLMs used in DevsecOps:

1. **Static Code Analysis of IaC:** Static code analysis of IaC generated configuration files has been explained in this section. The primary goal of [41] 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:** A study [42] proposes a machine learning-based approach using server log analysis to identify suspicious activities in run-time security using data connected to traffic. A novel technique using Python's ChatGPT utilizes context (labelled data and questions) and log entries to assess the indication of suspicious activities.

   The research focuses on identifying suspicious activity at run-time using machine learning predictive models against log records. The quantity of information needed to train a new prediction model each time when the log structure changes is one important consideration in this area. The research investigates and addresses the usage of ChatGPT's novel LLM for log analysis in the form of question-answer conversations with the objective of using lesser training data and adapting a pre-trained model for providing satisfactory results.

   The user's input represents a pair that is given as question and context where context means a group of labelled log entries which is sent to ChatGPT as sample data for pattern extraction. Based on the input, ChatGPT labels the network traffic record and also explains the meaning of the given log record, along with the underlying protocol, data exchange, and extra information pertaining to communication flow and network traffic. Finally, the user will be informed if there is any suspicious traffic or activity.

2.1.3 IaC Generation through ChatGPT Queries:

1. **ChatGPT for DevOps:** In this blog, [19] the author explained the usage of ChatGPT for DevOps for prompting the ChatGPT tool to produce small scripts in Python and Bash for which the results were accurate with code explanation. With the results generated, the author queried Chatgpt for some task-specific Terraform configuration, the configuration generated by the OpenAI Playground was identical to the manual configuration and comparatively better than the ChatGPT's result.

   The author continued with a few more prompts to ChatGPT with respect to DevOps concepts. Though the results were not so accurate and in fact similar to the Google search results, the usage of ChatGPT or OpenAI playground was helpful in understanding and getting a way out of some of the not only in CI/CD but even for debugging the code.

2. **SSO in Kubernates Configuration Generation:** In the blog [15], ChatGPT was used to generate SSO in Kubernates configuration in Azure, also kube-apiserver manifest details were queried for configuration for SSO using Azure Active Directory, it was seen that ChatGPT generates an example of a kube-API server YAML file and explains each parameter related to SSO. ChatGPT was queried to give an example of a kube-config file to use with the kube-apiserver, which it was able to generate. Next, it was asked to modify the kube-config by using oidc-login plugin of kubectl, which was able to give a proper solution.

   However, it is seen that when used as a solution to connect the oidc plugin with HTTPS to Azure, it gives an additional parameter "–https" which doesn't exist. When challenged, ChatGPT acknowledges this mistake and points to the direction of a reverse proxy, which is different from the actual desired parameters.

2.1.4 IaC Generation Tools with LLM

There are some tools which have been designed for generating IaC using LLMs, giving good results. An overview of these are given below:

1. **Infracopilot:** A cutting-edge IaC editor called InfraCopilot [35] is revolutionizing how cloud infrastructure is developed and managed. It uses Klotho[1], an open-source engine, that provides unparalleled intelligence, flexibility, and agility to create and change cloud architectures.

   Apart from the Klotho engine, The InfraCopilot service has several other components like API/Orchestrator, Intent Parser, Visualization Engine, and Discord Bot. To communicate requests to the service, the user engages with the Discord Bot. LLMs send the extracted user intent to the intent corrector, which confirms, corrects and converts it into a JSON format. The updated user intent is then expanded into a verified architecture by the Klotho Engine. All of the low-level

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

components, such as VPCs, subnets, security groups and IAM policies are a part of the multi-level architecture that the Klotho engine produces.

In contrast to other LLM design generators, InfraCopilot only uses LLM to comprehend user intent and Klotho Engine architecture to provide consistent, clear and reliable infrastructure creation, modifications and upgrades.

2. **K8sGPT** [53] is a tool that scans Kubernetes clusters for quick and effective diagnosis by providing the diagnosis results. Tool encodes SRE experience into its analysers for pertinent data extraction and AI enhancement.

   It includes built-in analyzers like PodAnalyzer, pvcAnalyzer, rsAnalyzer, serviceAnalyzer, eventAnalyzer and ingressAnalyzer for seamless troubleshooting in the Kubernetes cluster. The platform includes default analyzers and a few additional analyzers like hpaAnalyzer and pdbAnalyzer out of which few are activated automatically and customized for specific requirements. The variety of analyzers allows for efficient and streamlined cluster operation.

   Filters are used to control K8sGPT resource analysis. The command “k8sgpt filters list" is used to display a list of the available filters. Running the “k8sgpt filters add" command with a comma separation adds the multiple filters and “k8sgpt filters remove" removes them. “k8sgpt integration activate/deactivate" command with the name of the tool helps in the activating or deactivating the integration with other tools like Trivy. By indicating these integrations with the filter option, they can be included in the analysis.

3. **Pulumi AI**[2] introduced an AI Assistant to speed up the process of finding, comprehending and using the cloud infrastructure API for developing cloud infrastructure using LLMs and GPT.

   In order to enable intelligent resource identification and interaction within the Pulumi cloud, Pulumi Insights [26] provides IaC intelligence using generative AI and LLMs for enterprise, search and analytics spanning infrastructure and cloud.

   Additionally, Pulumi Insights offers teams crucial components for managing cloud footprints using AI and LLMs. With a Pulumi resource supergraph displaying metadata and linkages across cloud infrastructure, it facilitates learning, finding and constructing cloud architecture. Using their own data warehouse and BI tools, users can visualize Pulumi resource data for cost, compliance and operational use cases.

[2] **https://www.pulumi.com/ai/**

Chapter 3

Infrastructure as Code (IaC)

Infrastructure as code (IaC) manages computer data centers using machine-readable definition files, utilizing declarative approaches instead of interactive configuration tools or physical hardware setup. This method manages both physical and virtual computers and configuration resources. Infrastructure as Code is an infrastructure automation approach based on software development techniques, focusing on consistent, reproducible processes for system vision and modification, testing, and application.

3.1 Introduction

Compared to previous systems, cloud technologies make it much simpler to replace infrastructure. The benefits of virtualized resources, automated procedures, and a significant reduction in provisioning time can be found in cloud-based technology. However, it is challenging to lead, direct, and expand systems with unrestricted cloud technologies.

A new mindset must be formed in order to use fast-moving Cloud Age technologies in place of antiquated, slow-moving old methods. It is necessary to implement more rapid technological advancements to lower risk and boost quality. To do this, a fundamental shift in strategy and fresh perspectives on change and risk are necessary. A system management method from the Cloud Age known as "Infrastructure as Code" represents crucial change and is of the highest caliber and reliability.

When managing and deploying computer data centers, Infrastructure as Code (IaC) uses machine-readable definition files rather than interactive configuration tools or physical hardware setup.This procedure manages both physical and virtual computers, including bare-metal servers, as well as the configuration resources that go with them. There could be a version control mechanism for the definitions. Instead of maintaining the code through manual processes, the definition files code may use scripts or declarative definitions, but IaC more frequently uses declarative approaches.

IaC tools are categorized as below two:

1. **Provisioning:** Tools in this category provide infrastructure components for one or more cloud providers. Examples include HashiCorp's Terraform [23] and Pulumi[1].

[1] `https://github.com/pulumi/pulumi`

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

3.2 Pre-Iac and Phase-wise Evolution of IaC

Pre-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.

Evolution of IaC: Early approaches of IaC achieved automation and management by using configurable scripting languages like Shell script [18] etc. But these methods had limitations in version control, modularity, and ease of use. To overcome the issues in the previous phase, declarative configuration management tools were used for managing the infrastructure state although with certain limitations in scalability, flexibility and cloud support. With the rise in cloud computing technology and to fix the limitations of the previous phase, the need for dynamic and scalable infrastructure provisioning emerged by introducing infrastructure orchestration tools like Terraform, Ansible, Pulumi and on-demand provisioning through code to the cloud environment.

IaC by utilizing a descriptive model, is reliably versioned and deployed by consistently adhering to three fundamental steps:

1. Developers use Domain-Specific Language (DSL) [49] to specify the configuration state in a file.
2. The above-specified configuration file is transferred to a server, code repository or API.
3. The system determines how to configure based on the instructions by executing the file transferred to a server.

3.3 Approaches of IaC

There are two main approaches for IaC, Declarative (functional) and Imperative (procedural). The key difference between these two approaches is **“what"** *vs* **“how"**.

1. **Declarative:** Also recognized as the functional approach within the realm of Infrastructure as Code (IaC), embodies a paradigm shift in the orchestration and management of computing infrastructure. This methodology operates on the fundamental principle of defining the intended state of the infrastructure, relinquishing the need for explicit procedural instructions. Instead, the system autonomously undertakes the requisite steps to transition from the current state to the desired

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

configuration seamlessly. Examples of this approach include ***Terraform*** and ***AWS Cloudformation***, which are popular in the domain of infrastructure automation.

2. **Imperative:** Also called as the procedural approach, it stands as a pivotal paradigm within the landscape of IaC. Diverging from the declarative approach, Imperative IaC methodologies prescribe specific commands and procedural instructions to be executed in a sequential manner to achieve the desired configuration. This methodology emphasizes fine-grained control and precision, enabling operators to orchestrate infrastructure provisioning with meticulous attention to detail. Examples of this approach include ***Ansible*** and ***Pulumi***.

3.4 Core Concepts of IaC

1. **Declarative Configuration:** In IaC, infrastructure is defined declaratively, which means that we describe the desired state of our infrastructure rather than imperatively detailing the steps needed to reach that state. This approach abstracts away low-level implementation details, focusing on the end result.
2. **Version Control:** Infrastructure code, like application code, is saved and maintained in version control systems like Git. This allows for tracking changes over time, easy collaboration among team members, and the option to rollback to prior versions if necessary.
3. **Automation:** The provisioning and management of infrastructure is done automatically using IaC tools. Developers write code that defines the desired configuration, and the IaC tool takes care of the rest, carrying out the required actions to reach the desired state, saving them the trouble of manually configuring servers or networks.
4. **Immutable Infrastructure:** IaC advocates for the idea of immutable infrastructure, in which infrastructure components are never changed once they are deployed. Rather, all updates or modifications generate new instances of the infrastructure, increasing predictability and reliability.
5. **Orchestration and Dependency Management:** IaC tools often provide features for orchestrating complex deployment workflows and managing dependencies between different infrastructure components. This ensures that changes are applied in the correct order and that all dependencies are satisfied.

3.5 Infrastructure Coding Languages

Infrastructure coding languages, also known as Infrastructure as Code (IaC) languages, are programming languages specifically designed for defining and managing IT infrastructure in a code-based format. These languages enable developers and operations teams to automate the provisioning, configuration, and deployment of infrastructure resources such as servers, networks, and storage.

1. **Declarative Languages:** This play a pivotal role in infrastructure provisioning by enabling operators to specify the desired state of computing environments with precision and clarity. By abstracting away implementation logic and automating state management, these languages empower organizations to streamline the deployment process, enhance reproducibility, and ensure consistency across diverse environments. As organizations continue to embrace the principles of automation and DevOps, the adoption of declarative languages is poised to remain a cornerstone in the pursuit of agility, scalability, and operational excellence. Terraform, AWS Cloudformation, Azure Resource Manager (ARM) Templates, Google Cloud Deployment Manager, and Ansible are a few examples of declarative languages.

2. **Programmable, Imperative Infrastructure Languages:** It is possible to manage and deliver infrastructure resources using computer languages that are imperative and programmable. The deployment, configuration and management of infrastructure components like servers, networks, storage and more can be defined and automated using these languages. Declarative code is helpful for specifying the desired state of a system, especially when the expected results are fairly consistent. It's typical to specify the geometry of an infrastructure that you want to use often and consistently. However, there are occasions when you want to create reusable, transferable code that can result in many results based on the circumstance. Imperative, programmable code becomes crucial in this situation. Libraries and abstraction layers are better built using programmable, imperative languages, which also typically offer superior assistance for authoring, testing, and managing libraries. Terraform, Ansible, Puppet, and Chef also are well-known programmable, imperative infrastructure languages.

3. **Domain-Specific Infrastructure Languages**: A specific category of domain-specific language (DSL) called a domain-specific infrastructure language (DSIL) is intended for defining and managing infrastructure systems and components. A DSL is a computer language designed specifically for one area of application. A general-purpose language (GPL), on the other hand, is generally usable across domains.

   DSLs come in a variety of forms, such as scripting, graphical, and rule-based languages. The majority of DSILs are declarative, which means they specify what the system should do rather than how it should do it.

   When discussing DSILs, we frequently refer to scripting languages like Terraform's HashiCorp Configuration Language (HCL), as well as the YAML and JSON formats used in Kubernetes settings and AWS CloudFormation. These are all specialized languages that enable uniform, repeatable, and automated infrastructure description, management, and provisioning. The collaboration between development and operations teams is often made simpler by the readability and comprehension of DSILs. They permit infrastructure as code (IaC), which entails the definition and management of the infrastructure through the use of code, enabling version control, testing, and repeatability.

4. **General purpose languages**:You can use general-purpose programming languages for IaC even though there are a number of domain-specific languages (DSLs) established especially for it, such HashiCorp's HCL (used in Terraform) or AWS CloudFormation's JSON/YAML templates. Here are a few general-purpose languages that are frequently employed in IaC scenarios.

    - **Python**:Although the playbook specifications in the popular IaC tool Ansible use YAML, you may create complicated scripts or bespoke modules using Python. Python is a popular choice for scripting in IaC environments due to its straightforward syntax and extensive library selection.
    - **Ruby**: A Ruby-based DSL is used by Chef, another IaC application, for its cookbooks (recipes). This enables users to define their infrastructure using Ruby's full potential.
    - **Go**: For its configuration files, Google Cloud's Deployment Manager employs JINJA2 or YAML, however the development of specialized tools and plugins for controlling infrastructure typically uses Go. Additionally, Kubernetes, a well-known container orchestration technology, uses it as its preferred language.
    - **JavaScript/TypeScript**:With the help of the open-source IaC tool Pulumi, you can define and deploy cloud infrastructure using general-purpose languages like JavaScript and TypeScript.
    - **C#**:Additionally, Pulumi supports C#, enabling.NET developers to define and maintain infrastructure using well-known tools and syntax.
    - **Java**: Developers can define cloud resources using Java (as well as other languages like TypeScript and Python) by utilizing AWS CDK (Cloud Development Kit).

3.6 IaC Tools and Details

Few examples of Infrastructure as code(Iac) tools are as below

1. **Terraform:** Developed by **HashiCorp**, Terraform is an open source tool that enables you to define your infrastructure as code using a straightforward declarative language, deploy, and manage that infrastructure across numerous public cloud service providers (such as Amazon Web Services [AWS], Microsoft Azure, Google Cloud Platform, DigitalOcean), as well as private cloud and virtualization platforms (such as OpenStack, VMware), using just a few commands.

    Go is the programming language used to create Terraform. The Go code is reduced to a single binary (or, more precisely, a binary for each of the supported OS systems), which is appropriately named terraform.You don't need to run any additional infrastructure in order to utilize this binary to deploy infrastructure from your laptop, a build server, or pretty much any other computer. This is due to the fact that, invisibly, the terraform binary contacts one or more service providers, including AWS, Azure, Google Cloud, DigitalOcean, OpenStack, and others, on your behalf. As a result, Terraform will be able to use the API servers and infrastructure that those providers already run, as

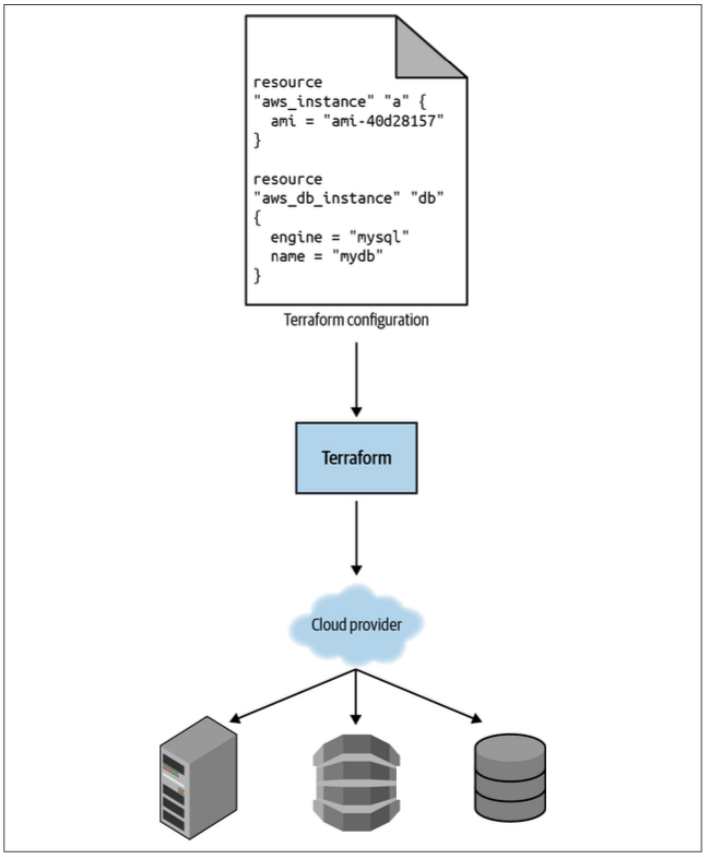

Figure 3.1: Terraform Provisioning on Multiple Providers

well as the authentication methods that you are already familiar with from those providers.(e.g., the API keys you already have for AWS).

How does Terraform know what API calls to make? The solution is to write text files called Terraform configurations, which describe the architecture you want to build. These setups make up the "code" in "infrastructure as code." Servers, databases, load balancers, network topology, and other components of your infrastructure may all be specified in Terraform configuration files, which you can then commit to version control. To deploy the infrastructure, you next issue specific Terraform commands, such as terraform apply. Your code is parsed by the terraform binary, which then converts it into a sequence of API calls to the cloud providers listed in the code. The terraform binary then executes these API calls as quickly as it can on your behalf.

Some key features of Terraform are as below

(a) **Platform Agnostic:** Numerous platforms are supported by Terraform. It offers a standardized syntax for expressing resources on virtually every cloud platform (including AWS, Google

Cloud, Azure, DigitalOcean, etc.), as well as on-premises with private clouds and direct hardware.

(b) **State Management:** In order to construct a strategy for achieving the infrastructure state you want and to ascertain what modifications are required to get there, Terraform keeps a state file.

(c) **Declarative Language:** The language used by Terraform is declarative. You specify your ideal condition, and Terraform figures out how to get there. Contrast this with a procedural language, where you must specify the precise actions to take in order to get the desired state.

(d) **Modular and Versioned:** The service maintains a versioned history of infrastructure changes and allows for the sharing and reuse of Terraform settings. This enables efficient teamwork and simple rollback.

(e) **Plan and Predict:**A user can examine and approve the actions before they are taken thanks to Terraform's ability to develop an execution plan that details what it will do to get to the target state.

Basic Terraform Configuration A basic Terraform Configuration contains 3 blocks as below:

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "example" {
  ami           = "ami-0c94855ba95c574c8"
  instance_type = "t2.micro"

  tags = {
    Name = "example-instance"
  }
}
```

Figure 3.2: Terraform Config

- **Provider: "aws":** Terraform is instructed to use the AWS provider by this block. In Terraform, providers are plugins that connect Terraform to a variety of services. Here, we indicate that we want to interact with AWS and that the region attribute for the area where we want to build resources should be us-west-2.
- **Resource: "aws_instance" "example":** A single resource that we want to manage is described in this resource block. Here, we're going to describe an EC2 instance in AWS.

example is the name we're giving this specific instance, and aws_instance is the resource type.

- Arguments for configuring the resource are contained within the resource block. Here, we're defining the instance type and the Amazon Machine Image (AMI) ID we wish to utilize for this instance.
- **Tags:** A map of string key-value pairs called tags corresponds to AWS tags. Giving our instance a name is all we're doing in this case.

You must use **terraform init** to initialize the Terraform working directory before executing this setup. Once that's done, use **terraform apply** to build the resource.

2. **Ansible:** Ansible is frequently referred to as a configuration management tool and is frequently discussed alongside Salt, Chef, and Puppet. When IT professionals discuss configuration management, they typically mean creating a state description for their servers and using a tool to verify that they are, in fact, in that state: the correct packages are installed, configuration files have the expected values and have the expected permissions, the correct services are running, and so on. Ansible offers a domain-specific language (DSL) that you use to define the status of your servers, similar to other configuration management systems.

   The deployment of software is another use for these tools. When people discuss deployment, they typically mean the process of creating binaries or static assets (if required) from software created by in-house developers, copying the necessary files to servers, adding configuration properties and environment variables, and launching services in a specific order. Some open source deployment tools include Capistrano and Fabric. Both deployment and configuration management are made easy with Ansible. The job of those in charge of system integration is made easier by using a single tool for both tasks.

   Some individuals discuss the necessity of orchestrating deployments. When several remote servers are involved in a deployment and certain steps must be taken in a precise order, orchestration is the process of coordinating deployment. For instance, you might need to start the database before starting the web servers, or you might need to remove the load balancer from the web servers one at a time to upgrade them without experiencing any downtime. Ansible is built from the ground up to be effective at this and to execute commands on several servers. It provides a paradigm for regulating the sequence of events that happens that is refreshingly straightforward.

   The topic of supplying new servers will finally come up. Creating fresh instances of virtual machines or cloud-native software as a service (SaaS) is referred to as provisioning in the context of public clouds like Amazon EC2. With modules for EC2, Azure, Digital Ocean, Google Compute Engine, Linode, Rackspace, and any other clouds that implement the OpenStack APIs, Ansible has you covered in this regard. Some key features of Ansible are as below

(a) **Agentless:** Ansible doesn't need an agent installed on the nodes it controls, unlike other configuration management technologies like Puppet or Chef. Your nodes are connected to, and "Ansible modules"—small programs—are pushed to them as part of the system's operation.

(b) **Idempotent:** Even if a task or operation is repeated numerous times, Ansible is made to ensure the same result. Idempotency, a feature of this kind, is essential for upholding the consistency and dependability of a system.

(c) **YAML Based:** Since YAML is simpler to read and write than other data formats like JSON or XML, Ansible utilizes it as its playbook language. A really straightforward configuration management and multi-machine deployment solution, unlike any that currently exist, and one that is very simple to use are based on a playbook.

(d) **Extensible and Flexible:** Users of Ansible can create custom modules and plugins. Additionally, it may be linked with numerous cloud computing infrastructures, such as AWS, Google Cloud, Azure, and others.

(e) **Security and Compliance:** Ansible assists in maintaining system consistency, lowering the possibility of human error, and assisting in the achievement of compliance objectives by automating common system administration operations.

(f) **Efficient Orchestration:** With the help of Ansible, complex multi-tier deployments may be managed because they are described in a playbook in plain, understandable language.

3. **Pulumi** is an open-source Infrastructure as Code (IaC) tool that allows to define and manage cloud infrastructure with the usage of programming languages like Java, Python, JavaScript, TypeScript, Go etc. instead of using Domain Specific Languages (DSL's) like YAML or JSON. *Below are the key features of Pulumi:*

   - **Programming Language Support:** It supports multiple programming languages to define infrastructure which helps in reusing the code and integration with existing tools.
   - **Declarative Infrastructure:** It allows user to define their infrastructure as code using a declarative style and provides high-level abstractions for common cloud resources, like virtual machines, databases, networks, and storage, which can be defined in a type-safe and effectively.
   - **Multi-Cloud and Hybrid Cloud:** It supports multiple cloud providers like Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and Kubernetes. Helps in defining our infrastructure and deploy it across different clouds or hybrid environments, to avoid vendor lock-in and simplify multi-cloud management.
   - **Continuous Delivery and Automation:** It integrates with popular Devops tools and practices, enabling us to incorporate infrastructure deployment into the CI/CD pipelines. Pulumi in conjunction with version control systems, such as Git, and continuous integration platforms

like Jenkins or Azure DevOps, helps in automating the deployment and management of the infrastructure.

- **Infrastructure as Software:** It helps in treating infrastructure code as software code such as for modularization, testing, and code reuse, to build and maintain your infrastructure. This also provides a rich ecosystem of libraries and modules, by allowing users to use community-contributed code and share infrastructure patterns with others.
- **State Management:** It manages the state of the infrastructure deployments, by storing it in a back-end system which enables Pulumi to perform updates to the infrastructure incrementally and maintain a record of the deployed resources that helps in managing updates, perform diffs, and track changes over time.

4. **Kubernetes** also known as **k8** or **Kube** is an open-source platform that automates the deployment, scaling, and management of containerized applications. It was originally developed by Google and maintained by the Cloud Native Computing Foundation (CNCF). *Few Key concepts of Kubernetes are mentioned below:*

    - **Pods:** A smallest unit in Kubernetes that represents a group of one or more containers deployed together on a single host. Containers within a pod share the same network namespace and can communicate with each other through localhost.
    - **Deployments:** Defines the desired state of a set of pods and manage their lifecycle. They provide features like scaling, rolling updates, and rollbacks and ensures that a specified number of replica pods are running and can automatically replace any terminated pods.
    - **Services:** This provides a stable network endpoint to access a group of pods and enables load balancing across multiple pod instances and provide service discovery within the Kubernetes cluster.
    - **ReplicaSets:** This is responsible for maintaining a specified number of replicas of a pod template. They are the predecessor to deployments and provide basic scaling and self-healing capabilities.
    - **Namespaces:** This provides a way to divide a Kubernetes cluster into virtual clusters, allowing multiple teams or projects to share the same physical infrastructure while isolating resources and policies.
    - **Ingress:** It is an API object which manages external access to services within a cluster by allowing us to define rules for routing external traffic to specific services based on paths or hostnames. This often works in conjunction with an Ingress Controller.
    - **Persistent Volumes and Volume Claims:** This provides a way to manage and abstract storage resources in Kubernetes. PV's represent physical storage, while PVC's are requests for specific storage resources. They allow you to dynamically provision and attach storage to pods, making it easier to manage data persistence.

- **ConfigMaps and Secrets:** This allows you to store and manage non-sensitive configuration data, such as environment variables or configuration files, as key-value pairs. Secrets are similar to ConfigMaps but are specifically designed for storing sensitive information, such as passwords, API keys, or TLS certificates. Both ConfigMaps and Secrets can be mounted as volumes or used as environment variables within pods.
- **Kubernetes API:** Kubernetes exposes a powerful API that enables programmatic management and automation of its resources. API can be accessed by using **'kubectl'** command-line, as well as through client libraries and SDKs in various programming languages.

3.7 Key Role and Objective of IaC in DevOps Practice

DevOps, characterized by enhanced collaboration between software development and IT operations teams, serves as a catalyst for streamlining the application development lifecycle and achieving continuous delivery of high-quality software. This paradigm shift necessitates the integration of IaC to automate infrastructure tasks seamlessly throughout the development process. By incorporating IaC into continuous integration and continuous deployment (CI/CD) pipelines, DevOps teams can synchronize infrastructure changes with software releases, facilitating rapid and reliable deployments.

1. **Streamlined Environment Setup:** DevOps teams leverage IaC to swiftly provision complete environments, spanning from development to production. By codifying infrastructure configurations, teams can automate the setup process, eliminating manual interventions and reducing setup time.
2. **Consistent Reproducibility:** IaC ensures that infrastructure configurations remain consistent and reproducible across different environments where the consistency measure minimizes the disparity between development, testing, and production environments by enhancing the reliability and predictability of deployments.
3. **Efficient Integration:** IaC tools seamlessly integrate with cloud providers, enabling DevOps teams to leverage cloud resources and services efficiently. By dynamically provisioning and scaling infrastructure resources based on demand, organizations can optimize resource utilization and reduce operational overhead.

3.8 Benifits of IaC

IaC offers numerous industry benefits [25, 11], 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.

1. **Automation and Consistency:** Automates infrastructure provisioning and management ensuring consistency and reproducible setups, reducing human errors and configuration drift.
2. **Agility and Speed:** Enables rapid and efficient deployment, with easy response to changing business needs and scale resources.
3. **Version Control and Transparency:** Enable teams to track changes, collaborate effectively, and roll back to previous versions and transparent for a team to understand the architecture.
4. **Scalability:** Automate the process of scaling the resource for efficient traffic management.
5. **Reusability and Modularity:** Its structure is modular and reusable by adapting across different environments.
6. **Immutable:** Changes are made by creating new instances without modifying the deployed instances to enhance stability and predictability.

3.9 Challenges of IaC

Though IaC offers the above benefits, it comes with its own set of challenges [52] such as an learning curve, increase in the complexity, consistency, compatibility, and dependencies across various platforms. It also involves expertise in handling scalability, proper version tracking, tool selection and security aspects. Below are also few major concerns:

1. **Learning Curve:** Adoption from manual approaches challenges teams in learning new tools, languages and practices.
2. **Complexity:** Complex configurations can lead to intricate code that requires careful design and testing.
3. **Tool Selection:** Selecting an appropriate tool for the specific task can be one of the difficulties because of various tools options that are open-sourced.

3.10 Conclusion

IaC have gained a remarkable traction as a paradigm that treats infrastructure as programmable code. With the increasing complexity and scale of modern infrastructure systems, the need for efficient and scalable approaches to infrastructure provisioning and management has become paramount. However, the manual creation of infrastructure code can be time-consuming, error-prone, and challenging to maintain, especially in complex environments. This is where Large Language models come into play, offering the potential to automate and streamline the generation of infrastructural code.

Chapter 4

Large Language Models (LLM)

This chapter demonstrates the evolution of Language Models (LM) through four vatious phases starting from statistical language models, progressing through Neural Language Models. Further this chapter also delves into the study of Lage Language Models (LLMs) and emphasises the advancements and need of LLMs. It also discusses about various LLM's like Code Generation models that are used as a part of this work.

4.1 Language Models

Language model (LM) is an AI model that is fundamentally designed to understand, generate and manipulate the human language. LMs have garnered substantial attention and transitioned through four distinct developmental stages starting from statistical language models, progressing through neural language models namely RNN [5, 36, 37], LSTM [21] and Transformers [59], further advancing into pre-trained language models like BERT [14] 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 and is a major breakthrough for LLMs. LLMs such as GPT's and RoBERTa [33] models exhibit high potential in performing NLP downstream tasks. The evolution of LM's is as follows:

1. **Rule-Based Approach:** In the early stages of NLP, it evolved with a rule based methods which is human written, a tedious and time-taking task. It had the advantages of easy interpretability, faster computation with less training data and high precision but it failed to cover few linguistic aspects and its laborious nature in creating the rules.

2. **Statistical-Based Approach:** Statistical LM (SLM), a method used for creating probabilistic models, proposed in early 1980's assigns the probabilities for sequence of N words and predicts following word using previous words. **Hidden Markov Model (HMM)**, **N-gram** are few popular models of this approach. Both the models are popularly used in Machine Translation, Time series analysis, POS-tagging tasks, text generation tasks.

- **HMM:** A probabilistic model that involves a sequence of observations, where each observation corresponds to a hidden state or set of probability distribution. Main objective of HMM is to understand and learn about the Markov chain by observing its hidden states.
- **N-gram:** Simple approach that predicts the probability distribution of 'n' words where 'n' can be any number and defines the size of the gram i.e sequence of words being assigned a probability. There are others too line Unigram, Bigram, Trigram which takes 1,2,3 grams respectively.

Limitations of SLM: SLM was very effective across various NLP tasks with good probabilistic interpretability and computationally faster models, it also had the limitations of data sparsity, limited context understanding, in handling words Out Of Vocabulary (OOV) and importantly curse of dimensionality i.e to handle large quantity data.

3. **Neural-Based Approach:** A neural network network based LM improved over n-gram model that helped in overcoming curse of dimensionality also by enabling easier learning of higher data and distributed representations.

   - **Reccurent Neural Network (RNN):** RNN, a successor for Feed-Forward Neural Network (FFNN) is a type of Artificial Neural Network (ANN) proposed by [5] in LM. It is claimed that RNN is able to handle the sequential and variable data due to its structure and parameter sharing. [36, 37] who proposed RNNLM affirms that unlike FFNN proposed by Bengio, RNN has no fixed window length and the connections between them helps the information to loop within the network layers using Backpropogation Throgh Time (BPTT). Though this model has performed better than FFNN and n-gram in terms of Perplexity (PPL), it has the challenges vanishing gradient that hinders learning long-term dependencies.
   - **Long Short-Term Memory (LSTM):** An RNN architecture with gradient-based learning approach for error back flow proposed by [21] overcomes the issue of vanishing or exploding gradients often encountered while training a traditional RNN model. Gating mechanism enables the LSTM's to learn the long-term dependencies by preserving useful information and discarding irrelevant or redundant information. The concept of using gates with its ability to handle the sequential data have made them one of the most successful architectures for a wide range of applications in NLP like machine translations, time series analysis.

4. **Transformers:** The transformer architecture [59] reduces reliance on recurrence and convolutions, using the concept of self-attention to capture long-term dependencies and parallelize computations effectively. This method impacted the field and became the core for LLM's like GPT, LLaMA and PaLM. It's architecture consists of the following components:

   (a) **Input Embedding:** An embedding layer that transforms each word in input sentence into continuous dense vectors, enabling the model to learn.

(b) **Positional Encoding:** A layer added to input embedding aid in understanding word placement in input by enabling the model to understand sentence structure without relying on word order, as self-attention does not account for word order.

(c) **Multilayered Encoder:** The encoder encodes sequential input using a multi-head self-attention module, using Transformer model with stack of identical layers dividing into a position-wise connected feed-forward network, self-attention mechanism with residual connection, and normalization step around each of two sub-layers.

(d) **Multilayered Decoder:** The decoder takes processed input, uses masked self-attention to output a sequence, using a stack of identical layers, two sub-layers as in encoder with an addition of third sub-layer for multi-head attention with each sub-layer surrounded by residual connections and normalization steps just like in encoder.

(e) **Final Linear and Softmax Layer:** This layer passes output of the decoder that generates probabilities for each word in the target vocabulary.

Popular loss function cross-entropy that calculates loss between the expected and actual output sequences, is used for training. For inference, the model generates one word at a time and feeds its previous predictions as input into the next step.

4.2 Large Language Models (LLM)

LLMs, an AI models trained on diverse internet text, generating coherent and context relevant sentences based on the input works on the principle of transformer architecture and self-attention mechanisms to handle long-range dependencies. Their size depends on parameter count.

They can perform tasks without task-specific training data, This includes tasks such as translation, question answering, and text generation. However they can produce biased, nonsensical, or misleading outputs. Large language models do not understand text as well as humans can, but they identify patterns in the trained data and use these patterns to generate responses. LLMs such as GPT's and RoBERTa [33] models exhibit high potential in performing NLP downstream tasks.

4.2.1 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 from [62], there are three ways to pre-train a code source 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 [7], a GPT-3 [6] 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[1] [58] trained on 25-30B tokens of Google BigQuery data, evaluated on HumanEval where CodeParrot 110M(small) outperformed the CodeParrot 2B(large). CodeGen [38], proposed Multi-Turn program synthesis model trained on The Pile [17], BigQuery[2] 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 [56, 57] 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 [3] 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 [39] uses Reinforcement Learning with Human Feedback [10, 54] along with models like code-davinci-002, text-davinci-003 that has shown their program synthesis abilities. PaLM [9] 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 [16] 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 [27] is another model that outperformed other models with lesser data and fewer epochs.

- **Encoder-Decoder Based LM:** This model utilizes both encoder-decoder blocks. CodeT5 [60] which extends T5 (Text-To-Text-Transfer-Transformer) [47] works on the objectives of masked span prediction, de-noising 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 [1] on all generation tasks.

4.2.2 Code Generation Models:

1. **Open-source models**

    - **CodeParrot:** A GPT-2 model trained for code generation task from scratch on Google BigQuery's python code with a collection of 20M files (each file of 1MB) with 180GB

[1] `https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot`
[2] Publicly available dataset released by Google `https://cloud.google.com/bigquery`

as the corpus size. As a part of data preprocessing, data is filtered on the rules proposed in OpenAI's Codex model [7] along with some set of new rules framed by the authors of CodeParrot as mentioned in this GitHub repository [3] and split into train and validation sets. A custom tokenizer efficient at code tokenization is created and trained the models with 110M & 1.5B on 16xA100 (40GB) machine with accelerate command for using multi-GPU with data parallel concept and mixed precision and evaluated on *Humaneval* dataset.

- **CodeGen:** A unique Multi-Turn program synthesis model proposed by [38] by adapting transformer model trained with 17.1 Billion parameters on natural language and programming language data is built sequentially on the 3 below datasets by applying pre-processing techniques like filtering, deduplicating, tokenising, shuffling and concatenating the data.
  - **ThePile:** Also called CodeGen-NL is a Natural language dataset of size 825.18 GiB collected by [17], from 22 diverse high-quality subsets, with Programming Language (PL) data collected from GitHub repositories with more than 100 stars that constitute 7.6% of the whole dataset for language modeling.
  - **BigQuery:** A Subset of Google's publicly available dataset also called CodeGen-Multi which consists of code in multiple (multi-lingual) programming languages i.e C, C++, Go, Java, JavaScript, Python for training.
  - **BigPython:** Monolingual dataset also called CodeGen-Mono contains data only in python by collecting only public, non-personal data from GitHub.
- **LLaMA models:** LLM models LLaMA & LLaMA-2 proposed by [56, 57] is based on the standard transformer architecture with certain proposed modifications like applying pre-normalization using RMSNorm [63], replacing ReLU with SwiGLU activation function [50], replacing positional encoding with rotary positional embeddings (RoPE) [55] on the top of transformer architecture. It's training process includes tokenization with vocab size of 32k using Byte-Pair Encoding algorithm (BPE) [48] implementated in SentencePiece [29], using AdamW optimizer [34] and cosine learning rate scheduler in both the LLaMa models. The only difference between the two models is the increase in the context length from 2k (LLaMA) to 4k (LLaMA-2) and adopted Grouped-Query Attention (GQA) [2]. Data that is publicly available, and compatible with open sourcing have been used for training LLaMA model. Along with the dataset used in LLaMA model, LLaMA-2 has been trained on a new mix of publicly available data woth 40% increase in the size of pre-training corpus. It is evaluated on tasks like common sense reasoning, mathematical reasoning, question answering and code generation. In code generation it evaluated the ability of the models to write code from a natural language description on the two benchmarks datasets HumanEval and MBPP [3].

2. **Closed-source models**

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

- **Generative Pre-trained Transformer (GPT):** A popular series of large-scale language models developed by OpenAI with the ability to generate human-like text based on the input it receives. **GPT-1** [45] is a 117 million parameter transformer model with only decoder block trained on BooksCorpus dataset for unsupervised language generation with an additional layer finetuned for specific tasks. **GPT-2** [46], a 1.5 billion parameters released in 2019, is trained on diverse internet text with transformer-based architecture. It eliminates task-specific input and fine-tuning stages, recognizing tasks based on input hints and performing tasks using output probabilities conditioned on input and task type i.e $p(output|input, task)$.
- **GPT-3:** GPT-3 [6], a significant 175 billion parameter AI model similar to GPT-2 architecture with increase in parameter size and sparse attention improving efficiency by attending a smaller subset of elements. Adaption of the concept of in-context learning from zero-shot that learns without any single sample to one-shot that learns with single sample and few-shot learning that learns in a more realistic, effective manner with few samples just like how human learns.
- **GPT-3.5:** A refined and 100x times improved version of GPT-3 also called as InstructGPT [39] with three variations i.e 1.3B, 6B and 175B parameters. It aims to reduce hazardous production to some extent by incorporating human value policies and Reinforcement Learning with Human Feedback [10, 54]. Models like code-davinci-002, a base model purely for code-based tasks, text-davinci-002, an InstructGPT model based on code davinci-002, text-davinci-003 and GPT-3.5-turbo a fine-tuned version of GPT-3.5 which is a highly famous ChatGPT dialogue model are few models of GPT3.5. GPT3.5-turbo performs well also for code and text generation tasks.
- **GPT-4:** The GPT-4 is a multimodal model that processes text and image inputs, producing text output. It consists of an encoder that converts the input into avector representation, decoder that converts vector representation into a text, and attention mechanism that enable both encoder and decoder to focus on the concentrate on important aspects of input and output. The key objective of the model is to enhance its ability to understand and produce natural language content in complex and nuanced contexts. It outperforms human test takers in various tests, with a score of top 10% in a mock bar exam. It has various applications, including transforming hand-drawn drawings into websites, extracting specific image from medical scanned images, and removing NSFW objects by classifying and identifying them.

- **Google PaLM:** Pathways Language Model(PaLM), a 540B parameter densely activated auto-regressive transformer model trained on 780B tokens from various filtered webpages, books, Wikipedia, news articles, source code, and social media conversations across 6144 TPU v4 chips. It is a standard transformer based architecture in a decoder only setup with few modifications like SwiGLU activation fuction over ReLU, usage of parallel formulation than serialised formulation in each transformer block which increases the training speed

by 15%, multi-query attention over k-attention heads, RoPE embeddings over positional embeddings, using shared input-output embeddings with no bias. Additionally they used SentencePiece vocabulary with 256k tokens and Adafactor optimizer [51] without factorization . They trained their model on 3 different tasks just for 1 epoch and evaluated on 29 English benchmarks like Pathways Language Model(PaLM) is a 540 billion parameter, densely activated, autoregressive Transformer on trained on 780 billion tokens of high-quality text. The Pathways (Barham et al., 2022) system, a new machine learning system that enables very efficient training of very large neural networks across thousands of accelerator chips, including those spanning several Tensor Processing Units (TPU) v4 Pods, was used to do this. In hundreds of tasks including natural language, coding, and mathematical reasoning, this new model, known as Pathways Language Model (PaLM), achieves cutting-edge few-shot results.

The key architectural element of PaLM model are:

(a) **SwiGLU Activation:** Since SwiGLU activations have been demonstrated to greatly improve quality in comparison to conventional ReLU, GeLU, or Swish activations, this was used for the MLP intermediate activations.

(b) **Parallel Layers:** Instead of the usual "serialized" formulation, a "parallel" formulation was adopted in each Transformer block. As a result of the ability to combine the MLP and Attention input matrix multiplications, the parallel formulation results in training times at large scales that are around 15% faster. It was estimated that the effect of parallel layers should be quality neutral at the 540B scale based on ablation studies that revealed a modest quality degradation at the 8B scale but no quality degradation at the 62B scale.

(c) **Multi-Query Attention:** The input vector is linearly projected into "query", "key", and "value" tensors of shape $[k, h]$ for each timestep in the typical Transformer formulation, which employs k attention heads. h is the size of the attention head. Since "key" and "value" are projected to $[1, h]$ in this case, but "query" is still projected to shape $[k, h]$, the key/value projections are shared for each head. According to research, this has a minimal impact on model quality and training efficiency, but it significantly reduces the cost of autoregressive decoding. This is due to the fact that only one token can be decoded at a time, the key/value tensors are not shared between samples, and normal multi-headed attention performs poorly on accelerator hardware during auto-regressive decoding.

(d) **RoPE Embeddings:** Since RoPE embeddings have been demonstrated to function better on extended sequence lengths, this is utilized instead of absolute or relative position embeddings.

(e) **Shared Input-Output Embeddings:** Input and output embedding matrices are shared, which has been done previously frequently (but not always).

(f) **No Biases:** None of the dense kernels or layer norms used any biases. For large models, it was discovered that this approach boosted training stability.

(g) **Vocabulary:** The training corpus contains a significant number of languages, thus a SentencePiece vocabulary of 256k tokens was chosen to support this diversity. This vocabulary was created from the training data, which increases training effectiveness.

- **Google PaLM2/BARD:** The language model supporting Google's chatbot Bard is PaLM 2. PaLM 2 is a language model that succeeds PaLM and integrates modeling advancements, data improvements, and scaling insights. The main advancements of PaLM2 as as below:

  (a) **Compute-Optimal Scaling:** Compute-optimal scaling recently demonstrated that data size is at least equally significant to model size. In order to attain the highest performance for a given quantity of training compute, this study is validated for greater amounts of compute and discover that data and model size should be scaled about 1:1 (as opposed to previous trends, which scaled the model 3 faster than the dataset)

  (b) **Improved Dataset Mixtures:** English text made up the majority of previous big pre-trained language models' datasets. PaLM2's pre-training mixture spans hundreds of languages and areas (such as computer languages, mathematics, and parallel multilingual papers), and is more multilingual and diversified. PaLM2 demonstartes how deduplication can be used to reduce memorization and how larger models can handle more diverse non-English datasets without affecting English language understanding performance.

  (c) **Architectural and Objective Improvements:** The Transformer serves as the basis for the model architecture. A single causal or masked language modeling objective has been employed nearly entirely in previous LLMs. A tuned blend of various pre-training objectives was used in this model to train it to grasp various facets of language in light of the strong results of UL2 [12]

  (d) **Coding Support:** A compact, coding-specific PaLM 2 model was developed by continuing to train the PaLM 2-S model on a large, code-heavy, substantially multilingual data mixture for low-latency, high-throughput deployment in developer workflows. The resulting model, which is referred to as PaLM 2-S*, significantly improves on code tasks while maintaining performance on natural language tasks.

Chapter 5

IaC using LLMs: Training and Evaluation

This chapter explores the application of Large Language Models (LLMs) in generating Infrastructure as Code (IaC), with a focus on Terraform. It underscores the importance of pre-trained models in Natural Language Processing (NLP) for various tasks, particularly text generation. Additionally, the chapter delves into the concept of in-context learning within LLMs, known as "Zero-shot" or "Few-shot" learning, demonstrating its relevance in swiftly inferring expected behaviors. Through this exploration, it aims to demonstrate how LLMs can streamline infrastructure management workflows, especially in IaC generation using Terraform.

5.1 Model Training

5.1.1 Pre-Training

Pre-trained models [44] in the NLP serves as the fundamental building blocks for a wide range of downstream tasks, each involving diverse data modalities. This technique trains the model using large benchmark data and tasks for easy fine-tuning in various applications that make it easier to solve new tasks. This enables well-trained LMs to capture and learn the rich knowledge and semantic representation by utilizing unlimited training data from unlabeled text corpus for downstream tasks like text generation etc.

5.1.2 Training GPT 3.5-Turbo using In-Context Learning

- **In-Context Learning:** In-context learning in this context 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 ***“a 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 a “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.

A model with a few-shot setting in LLM like GPT 3.5-Turbo for generating a "text to terraform configuration", uses 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.

- **“Hard" prompt tuning:** Improve prompt performance by feeding many variants of prompts and selecting optimal prompt formulation without changing model parameters using smaller labelled datasets.

- **Indexing:** An in-context learning-based technique transforms LLMs into information retrieval systems to extract data from various sources. It involves a vector database parsed into smaller pieces by indexing to calculate the similarity between query embeddings and each vector by helping to retrieve top-K embeddings to synthesize the response.

5.1.3 Fine-tuning Code-Parrot

- **Instruction fine-tuning:** LLMs prompted for generating IaCs might exhibit unanticipated behaviours, such as fabricating information or generating harmful content, due to their current target of “Generating IaC with an IaC prompt" being out of alignment with the pre-trained language modelling objective of “Predicting the next token"

  Therefore, it is essential to be fine-tuned for these tasks. The process of fine-tuning involves identifying the task for designing the ideal architecture, loss function, and data selection. Thus our work, Generating “Infrastructure as Code" configuration files from English text will be a powerful tool for engineers to refine configurations after the review.

  The pre-trained model learns linguistic features from vast code and then the text corpus must be loaded and can be adapted for fine-tuning. The adapted model for fine-tuning entails a continuous training process on task-specific data, in our case the terraform prompts and configuration examples. We keep tuning the pre-trained parameters of the model to better suit our task and use gradient descent for updating parameters on task-specific data loss. Performance is improved by adjusting hyper-parameters such as training epochs, batch size, learning rate, and weight decay. The improved model is kept for analysis. Code-parrot fine-tuning was done for 20000 epochs with the datasets mentioned in the section section 5.2.

- **Parameter Efficient Fine-Tuning(PEFT):** LLM adjustment requires a substantial amount of computational resources due to the scale of the models which might be a challenge for developers with limited computational resources.

  Parameter-efficient Fine-Tuning(PEFT), a technique developed in recent times effectively adapts pre-trained transformers that improve model performance, and efficiency by fine-tuning on a small set of parameters, memory utilization [32], training speed, model excellence, and inference expenses. The important PEFT techniques are as below:

  1. **Reparameterization:** These techniques reparameterize network weights using low-rank transformation, reducing trainable parameters while maintaining the method's ability to operate on high-dimensional matrices, such as the pre-trained network parameters. Low-rank adaptation (LoRA) [24] is a popular Reparameterization method. Low-rank adaptation refers to reparameterizing pre-trained LLM weights using low-rank transformations. QLoRA [13] is another efficient fine-tuning approach that reduces memory usage enough to fine-tune a 65B parameter model on a single 48GB GPU while preserving full 16-bit fine-tuning task performance. QLORA backpropagates gradients through a frozen, 4-bit quantized pre-trained language model into Low-Rank Adapters (LoRA).
  2. **Additive methods:** Additive approaches enhance pre-trained models with additional parameters or layers, forming the most researched group of PEFT techniques, including soft prompts and adapter-like approaches.

     Adapters methods [22] add additional parameters to the transformer layers. In the initial proposal, additional fully connected layers were added after the multi-head self-attention and existing fully connected layers in each transformer block. During LLM training via the adapter approach, solely the new adapter layers are updated, keeping other transformer layers fixed.

     Adding a trainable parameter tensor (the "soft prompt") to the embedded question tokens is the primary concept of soft prompt tuning [30]. Gradient descent is then used to modify the prepended tensor to enhance modelling performance on a target dataset.

     "Prefix tuning"[31], differs from soft prompt tuning in a way that trainable tensors (soft prompts) are prepended to each transform block rather than only the embedding inputs.

5.2 Dataset Description

The major step of the work is to collect the data specific to the task. To generate IaC for Terraform, we collected the data by running a query on Google BigQuery GitHub dataset with a .tf extension, which contains over 164MB of Terraform data i.e. 23839 files out of 1.5TB of the corpus. Required pre-processing steps like removing duplicate files, tokenising and divide into training and validation sets

in a 75/25 ratio. For our sample example with Code-parrot 19071 files were used for training and 4768 files for validation.

5.3 Model Evalution

In order to benchmark generative models for code, samples are typically compared to a reference solution for Functional correctness [7], the match may be exact or non-exact (as in BLEU/ROUGE score).

Matching generated and reference configurations employs three distinct methods as below mentioned. For our work we utilized Functional Correctness by Exact Match for IaC evaluation.

- **BLEU:** By comparing the execution plan of the generated configuration file to the required configuration, BLEU [40] determines the average precision over a range of n-gram sizes.
- **Rouge:** variants of Rouge[8] like ROUGE-1, ROUGE-2, ROUGE-L, Recall, Precision, and F1 scores can be used to measure the match between configuration file execution plans.
- **Functional Correctness by Exact Match:** This metric compares the function of a generated configuration file to a reference solution. It also ensures that different setups but the same functionality are taken into account. The LLM-generated configuration file will be used by Terraform to develop an JSON execution plan. A JSON file containing the plan is created if the generated configuration file compiles, and if it matches the reference solution plan, the generated configuration file is deemed successful. This implies that even the slightest error in the configuration file generated will be considered 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.

The evaluation dataset's purpose is to cover all features of Terraform configuration files that includes jobs from GCP, AWS, and Azure, as well as infrastructure ranging from virtual machines to VPNs. The complexity of The task varies widely in order to push the LLMs to their maximum capability. The identities, names, descriptions, and timestamps should be deleted from the JSON files before the two plans are compared. As code or configuration generation is stochastic, for each task multiple samples should be generated to determine the average success rate.

The identities, names, descriptions, and timestamps should be deleted from the JSON files before the two plans are compared. As code or configuration generation is stochastic, for each task multiple samples should be generated to determine the average success rate.

5.4 Experiment Details

As part of our work, experiments were conducted using four Nvidia GeForce RTX 2080 Ti GPUs, each equipped with 11GB of memory. The models evaluated in these experiments include the 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. Two key hyperparameters, namely temperature and number of samples, were manipulated to assess the model's performance. ***Temperature*** is a hyper-parameter that influences the text's creativity and diversity. Lower value is more deterministic and focused in the generation while higher value is more diverse and creative.

The Table 5.1 summarizes the results obtained by configuration generation and evaluation by functional correctness by exact match with human generated Terraform configuration for two models.

1. **GPT 3.5-Turbo:** In this model we used in-context learning for the model to generate the configuration for all 49 tasks in AWS provider. 1-sample and 50-sample configurations were generated for each of the 49 tasks using two temperatures 0.2 and 0.6. These configurations were evaluated with human generated configurations by an functional correctness criterion. As can seen from the results, 59.16% accuracy was obtained with 1-sample and 56.81% accuracy was obtained by aggregating over 50-samples

2. **CodeParrot:** This model is used to generate 1-sample for each of the tasks and compared all 49 tasks with the human Terraform configurations with the exact match procedure outlined in the evaluation section above and measured the average success rate. Similarly, the same approach was done with 50-samples for each of the 49 tasks and the average success rate was calculated. As can seen from the results table, 8.2% accuracy was obtained with 1-sample and 8% accuracy was obtained by aggregating over 50-samples.

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

Table 5.1: Experimental Results

5.5 Results Analysis

In summary, GPT 3.5-Turbo outperforms the CodeParrot model. This superior performance can be attributed to several factors, including its extensive and diverse training dataset, as well as its inherent adaptability through fine-tuning. These aspects collectively contribute to GPT 3.5-Turbo's remarkable performance compared to CodeParrot.

The reason for CodeParrot's underperformance is due to the Model's nature and specificity, as this model is trained on vide code data and doesn't have the specific domain knowledge and understanding.

Also, the another analysis from this work says that the model has been working efficiently on simple tasks i.e., those involving the provider block and one small resource block by failing in generating large configurations. Along with these, we also observed that the model is getting stuck in one block and generating the same text over and over again.

Ex: In the **aws_security_group resource**, it set **vpc_id = "$aws_vpc.vpc-web.id"** even though that vpc resource did not exist.

Chapter 6

IaC using LLM: Safety and Ethical Considerations

Infrastructure as Code generated by LLMs have some challenges wrt the accuracy of the configurations generated and safety of using these in production environments. This chapter outlines the safety and ethical considerations for generation of IAC using LLMs and possible resolutions

6.1 Safety Concerns:

- **Security Risks:** Unvalidated IaC can lead to vulnerabilities like misconfigured databases, lax security, and exposed secrets. 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.

6.2 Best Practices:

- **Review and Validate:** Continuous evaluation of IaC for performance, security, and compliance through manual and automated reviews.
- **Test in Isolated Environments:** Pre-deployment 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.

6.3 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.

6.4 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.

In conclusion, LLMs have the ability to generate IaC with great efficiency, but they must be employed carefully and with awareness of their ethical ramifications.

Chapter 7

Conclusions and Future work

Leveraging Large Language Models (LLMs) for Infrastructure as Code (IaC) generation presents significant challenges. AI training using GitHub's limited dataset, Terraform accounting for just a fraction of 500,000 HCL repositories, may yield syntactically sound yet erroneous code for intricate infrastructures. IaC entails handling sensitive data, revealing LLMs' ignorance of the latest practices, vulnerabilities and posing security risks. Lack of context in LLMs risks offering ill-fitting standard solutions, misaligning with diverse use cases and affecting infrastructure needs. Cloud provider and tool API updates render LLM-generated code suboptimal for specific features. Testing intricate LLM-generated IaC intensifies deployment complexities, while the absence of best practices and comments hampers modifications. Integration issues with DevOps tools arise if LLM code significantly differs from manual IaC. Inefficient LLM-generated IaC inflates costs due to suboptimal resource usage.

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.

7.1 Future Work

This section outlines certain future research possibilities of IaC with LLMs.

Our analysis has inspired a new direction of work focused on building and fine-tuning an exclusive task specific model capable of generating Infrastructure as Code (IaC) configurations based on input context as there is no existing model publicly available for this specific task.

Additionally, this research has sparked interest in exploring translation problems within the domain of cloud computing, specifically translating configurations between different cloud platforms (e.g., AWS to Azure or vice versa). This initiative of work presents exciting opportunities for applying techniques from natural language processing to infrastructure management by advancing the concept of Generative AI for practical engineering tasks.

1. **Expanding the Dataset:** Expanding datasets with 1000 samples/tasks and performing a comparative analysis with open and closed-source models to maximize performance across various models by making it task efficient.

2. **IaC Chatbots:** In future, we can also use LLMs as assistants in multi-turn IaC Chatbots with automatic validations.

3. **Advanced Troubleshooting and Debugging:** Implement an LLM powered systems that analyze infrastructure configurations data, identify the issues, recommend the solutions, and autonomously fix problems without human intervention. This needs a lot of training on vast troubleshooting scenarios datasets enhances diagnostic capabilities.

4. **Cost Optimization and Management of Compute Setup:** Utilize LLM's to analyze infrastructural setup thet in turn recommends cost-saving measures, enhance performance, and optimize resource usage.

Related Publications

- **Kalahasti Ganesh Srivatsa**, Sabyasachi Mukhopadhyay, Ganesh Katrapati and Manish Shrivastava. **"A Survey of using Large Language Models for Generating Infrastructure as Code"**. In Proceedings of the 20th International Conference on Natural Language Processing (ICON), 2023, India. Association for Computational Linguistics.

Bibliography

[1] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[2] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[3] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[4] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[5] Y. Bengio, R. Ducharme, and P. Vincent. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

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

[8] L. Chin-Yew. Rouge: A package for automatic evaluation of summaries. In *Proceedings of the Workshop on Text Summarization Branches Out, 2004*, 2004.

[9] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[10] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[11] C. A. Cois, J. Yankel, and A. Connell. Modern devops: Optimizing software development through effective system interactions. In *2014 IEEE international professional communication conference (IPCC)*, pages 1–7. IEEE, 2014.

[12] M. Dehghani, J. Djolonga, B. Mustafa, P. Padlewski, J. Heek, J. Gilmer, A. P. Steiner, M. Caron, R. Geirhos, I. Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. In *International Conference on Machine Learning*, pages 7480–7512. PMLR, 2023.

[13] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] B. Entzmann. Chatgpt vs devops. `https://www.dbi-services.com/blog/chatgpt-vs-devops/`, 2023. Accessed: 2023-02-01.

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

[17] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[18] M. Greenberg, K. Kallas, and N. Vasilakis. Unix shell programming: the next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 104–111, 2021.

[19] C. Harvey. Using chatgpt for devops. `https://blog.devgenius.io/using-chatgpt-for-devops-7daa7c1783e9`, 2023. Accessed: 2023-02-09.

[20] M. Heap and M. Heap. Advanced ansible. *Ansible: From Beginner to Pro*, pages 137–157, 2016.

[21] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[22] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[23] M. Howard. Terraform–automating infrastructure as a service. *arXiv preprint arXiv:2205.10676*, 2022.

[24] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[25] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[26] insights. Pulumi insights: Intelligence for cloud infrastructure. `https://genesis-aka.net/information-technology/professional/2023/04/19/pulumi-insights-ai-generated-iac-programs/`, 2023. Accessed: 2023-04-13.

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

[28] M. Kawaguchi, K. Mizutani, and N. Iguchi. An implementation of misconfiguration prevention system using language model for a network automation tool. *IEICE Proceedings Series*, 72(S5-8), 2022.

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

[30] B. Lester, R. Al-Rfou, and N. Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[31] X. L. Li and P. Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[32] V. Lialin, V. Deshpande, and A. Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 2023.

[33] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[34] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[35] C. Masolo. Infracopilot, a conversational infrastructure-as-code editor. `https://www.infoq.com/news/2023/05/Infracopilot-conversation-editor/`, 2023. Accessed: 2023-05-31.

[36] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, pages 1045–1048. Makuhari, 2010.

[37] T. Mikolov, S. Kombrink, L. Burget, J. Černockỳ, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5528–5531. IEEE, 2011.

[38] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[39] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[40] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. 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, 2002.

[41] N. Petrović. Chatgpt-based design-time devsecops. *preprint*, 2023.

[42] N. Petrović. Machine learning-based run-time devsecops: Chatgpt against traditional approach. *preprint*, pages 1–5, 2023.

[43] S. Pujar, L. Buratti, X. Guo, N. Dupuis, B. Lewis, S. Suneja, A. Sood, G. Nalawade, M. Jones, A. Morari, et al. Automated code generation for information technology tasks in yaml through large language models. *arXiv preprint arXiv:2305.02783*, 2023.

[44] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020.

[45] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[46] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[47] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[48] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

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

[50] N. Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[51] N. Shazeer and M. Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[52] C. Siebra, R. Lacerda, J. Peixoto, I. Cerqueira, F. Da Silva, and A. Medeiros. 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, 2018.

[53] J. Singh. Unlocking the power of kubernetes with k8sgpt. `https://medium.com/google-cloud/unlocking-the-power-of-kubernetes-with-k8sgpt-c9b82d6ef205`, 2023. Accessed: 2023-06-18.

[54] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[55] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[56] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[57] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[58] L. Tunstall, L. Von Werra, and T. Wolf. *Natural Language Processing with Transformers, Revised Edition*. O'Reilly Media, Incorporated, 2022.

[59] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[60] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[61] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. 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, 2020.

[62] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[63] B. Zhang and R. Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.